Introduction

This guide provides information about application interfaces to help you select and implement functions for your OS/2 multimedia applications. It is written for application programmers who are interested in developing OS/2 multimedia applications.

Additional Multimedia Information

Developing Multimedia Applications Under OS/2 , John Wiley & Sons, Inc., 1995. (ISBN 0-471-13168-7)
Provides insight, tips and techniques, and samples on MMPM/2 application development. It also describes the uses of multimedia for general Presentation Manager applications.

Guide to Multimedia User Interface Design , IBM Corporation, 1992. (S41G-2922-00)
Describes design concepts to be considered when designing a CUA multimedia interface that is consistent within a particular multimedia product and across other products.

IBM Redbooks
You can find information on multimedia-related IBM Redbooks at http://www.redbooks.ibm.com/redbooks/ on the World Wide Web.

Multimedia with REXX - (online)
Describes REXX functions that enable media control interface string commands to be sent from an OS/2 command file to control multimedia devices. This online book is provided with the OS/2 operating system and is located in the Reference and Commands folder.

Using This Online Book

Before you begin to use this online book, it would be helpful to understand how you can:

- Expand the Contents to see all available topics
- Obtain additional information for a highlighted word or phrase
- Use action bar choices.

How To Use the Contents

When the Contents window first appears, some topics have a plus (+) sign beside them. The plus sign indicates that additional topics are available.

To expand the Contents if you are using a mouse, select the plus sign (+). If you are using a keyboard, use the Up or Down Arrow key to highlight the topic, and press the plus key (+).

To view a topic, double-click on the topic (or press the Up or Down Arrow key to highlight the topic, and then press Enter).

How To Obtain Additional Information

After you select a topic, the information for that topic appears in a window. Highlighted words or phrases indicate that additional information is available. You will notice that certain words in the following paragraph are highlighted in green letters, or in white letters on a black background. These are called hypertext terms. If you are using a mouse, double-click on the highlighted word. If you are using a keyboard, press the Tab key to move to the highlighted word, and then press the Enter key. Additional information will appear in a window.

How To Use Action Bar Choices

Several choices are available for managing information presented in the M-Control Program/2 Programming Reference. There are three pull-down menus on the action bar: the **Services** menu, the **Options** menu, and the **Help** menu.

The actions that are selectable from the **Services** menu operate on the active window currently displayed on the screen. These actions include the following:

Bookmark

Sets a place holder so you can retrieve information of interest to you.

When you place a bookmark on a topic, it is added to a list of bookmarks you have previously set. You can view the list, and you can remove one or all bookmarks from the list. If you have not set any bookmarks, the list is empty.

To set a bookmark, do the following:

1. Select a topic from the Contents.
2. When that topic appears, choose the **Bookmark** option from the **Services** menu.
3. If you want to change the name used for the bookmark, type the new name in the field.
4. Select the **Place** radio button (or press the Up or Down Arrow key to select it).
5. Select **OK**. The bookmark is then added to the bookmark list.

Search

Finds occurrences of a word or phrase in the current topic, selected topics, or all topics.

You can specify a word or phrase to be searched. You can also limit the search to a set of topics by first marking the topics in the Contents list.

To search for a word or phrase in all topics, do the following:

1. Choose the **Search** option from the **Services** pull-down.
2. Type the word or words to be searched.
3. Select **All sections**.
4. Select **Search** to begin the search.
5. The list of topics where the word or phrase appears is displayed.

Print

Prints one or more topics. You can also print a set of topics by first marking the topics in the Contents list.

You can print one or more topics. You can also print a set of topics by first marking the topics on the Contents list.

To print the document Contents list, do the following:

1. Select **Print** from the **Services** menu.
2. Select **Contents**.
3. Select **Print**.
4. The Contents list is printed on your printer.

Copy

Copies a topic you are viewing to a file you can edit.

You can copy a topic you are viewing into a temporary file named TEXT.TMP. You can later edit that file by using an editor such as the System Editor.

To copy a topic, do the following:

1. Expand the Contents list and select a topic.
2. When the topic appears, select **Copy to file** from the **Services** menu.

The system copies the text pertaining to that topic into the temporary TEXT.TMP file.

For information on any of the other choices in the **Services** menu, highlight the choice and press the F1 key.

Options

Changes the way the Contents is displayed.

You can control the appearance of the Contents list.

To expand the Contents and show all levels for all topics, select **Expand all** from the **Options** menu.

For information on any of the other choices in the **Options** menu, highlight the choice and press the F1 key.

What's New...

This release of the *OS/2 Multimedia Application Programming Guide* includes the following:

- Using a Control Program Queue for Notifications

  An OS/2 application that does not have a PM window, and therefore cannot use a PM message queue for receiving notification messages, can use an OS/2 Control Program queue instead. The Control Program queue method of notification should also be considered for time-critical PM applications, because it is faster than using the PM message queue.

- Direct Audio RouTines (DART) Interface

  The DART interface enables games and multimedia applications to bypass the waveaudio device entirely and communicate directly with the amp mixer.

- Additional DIVE functionality

  DIVE transparent blitting functions enable an interactive game or imaging application to create composites of graphics and image data using a transparency key color. See Transparent Blitting for more information.

- Additional flags for MCI_CUE

  Additional MCI_CUE flags allow digital video devices to seek to a specified position and to display or hide the video window when cueing the media. See Playing Motion Video Files for more information.

Multimedia Application Programming Environment

OS/2 multimedia (referred to as MMPM/2 or Multimedia Presentation Manager/2 in previous releases) is the multimedia platform for today because it takes advantage of OS/2 features to provide an effective multimedia environment. OS/2 multitasking capability supports synchronization and concurrent playback of multiple devices. The flat memory model supports the management of large data objects.

OS/2 multimedia is also the multimedia platform for tomorrow because of its extendable architecture, which enables new functions, devices, and multimedia data types to be added as the technology of multimedia advances.

Because OS/2 multimedia and its devices are designed to support synchronization activities, Presentation Manager (PM) applications can easily incorporate multimedia function for playing multiple devices concurrently and synchronizing audio and video as media drivers become available.

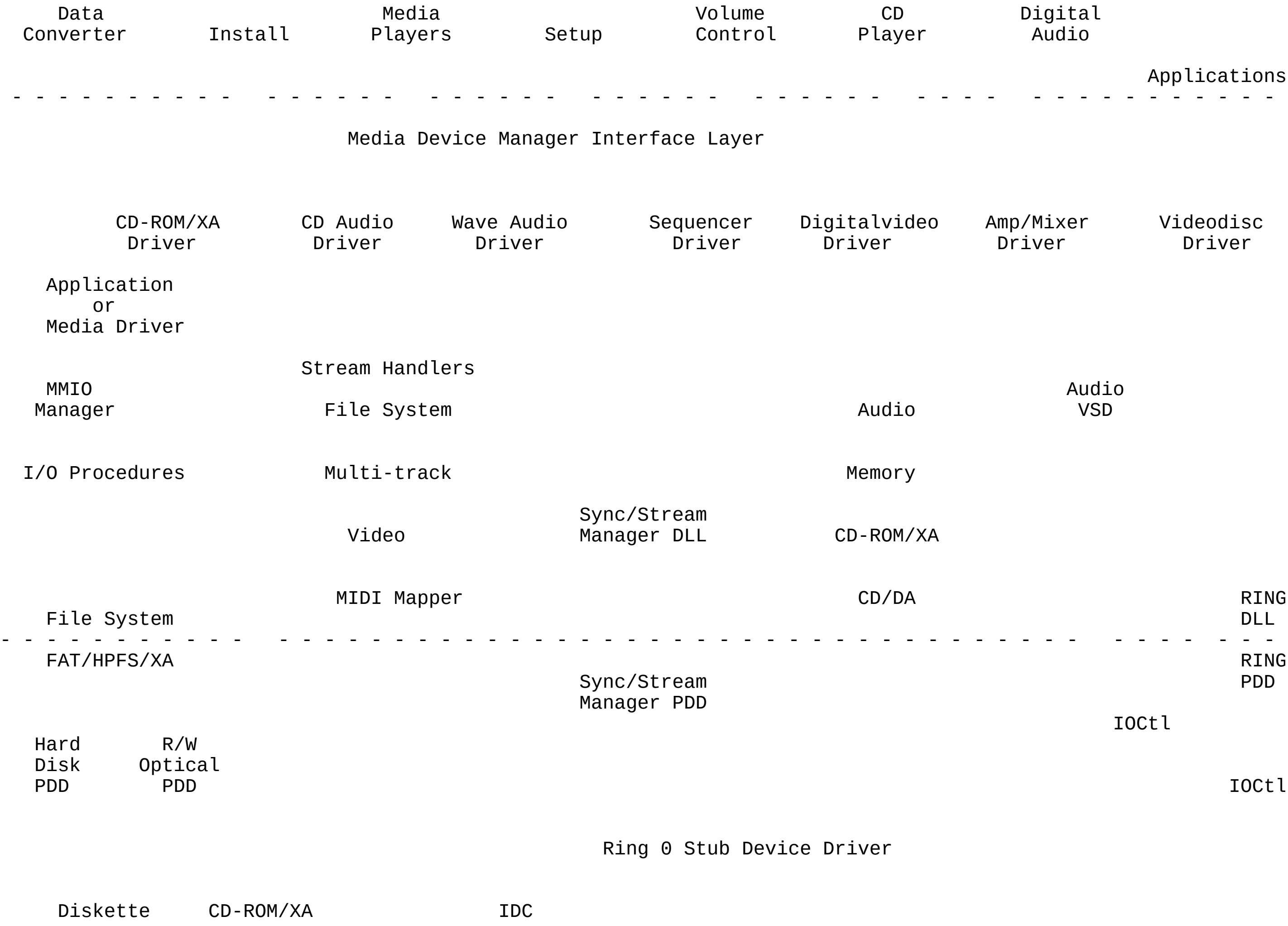
Application Programming Model

The application programming model for an OS/2 multimedia application is an extension of the OS/2 Presentation Manager programming model, providing both messaging and procedural programming interfaces. OS/2 multimedia API procedures allow applications to manage data and control devices while messages from the OS/2 multimedia system notify applications of asynchronous events.

The media control interface provides a view of the OS/2 multimedia system to both application developers and users that is similar to that of a video and audio home entertainment system. Operations are performed by controlling the processors of media information, known as *media devices* . Media devices can be internal or external hardware devices, or they can be software libraries that perform a defined set of operations by manipulating lower-level hardware components and system software functions.

Multiple media devices can be used in an operation. For example, the playback of an audio compact disc can be implemented by coordinating the control of a compact disc player and an amplifier-mixer device.

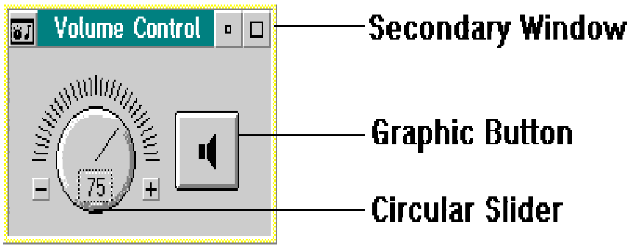
The Media Device Manager (MDM) shown in the following figure provides resource management for media devices and enables the command message and command string interface. The Media Device Manager provides device independence to an application developer.



```
        PDD          PDD                             Common Audio Interface

                    Generic SCSC                               OEM Audio                                  OS/2
                         DD                                      Device                                    COM
                                                                 Driver                                    PDD

- - -   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -   - - -

                                                                                                      RS-232
  Hard      Diskette  Optical   CD-ROM                      OEM Audio
  Drive      Drive     Drive     Drive                                                                        Physi
                                                                                                          Devic
```

Refer to the *OS/2 Multimedia Subsystem Programming Guide* for information on multimedia subsystem programming including media drivers and stream handlers.

To assist you in creating a standardized user interface for your OS/2 multimedia application, OS/2 provides multimedia window controls, which have been implemented in OS/2 multimedia applications such as *Volume Control*. See the following figure.



Graphic buttons are two-state buttons that can be toggled up and down. They can display text, or graphics, or both. They can also be animated. Their versatility makes graphic buttons ideal to use for device control panels.

Circular sliders lend realism to your panel by providing familiar-looking dials. The dials are easy to operate and do not hog screen real estate.

Secondary windows provide a sizeable dialog window to contain your multimedia device controls.

OS/2 Multimedia Application Requirements

The IBM Developer's Toolkit for OS/2 Warp includes the bindings, header files, and libraries for developing OS/2 multimedia applications.

A PM message queue is required for all OS/2 multimedia applications because it enables the efficient sharing of devices in the OS/2 multimedia environment.

The minimum stack size for an OS/2 multimedia application is 32KB; however, for better performance, use a stack size of 64KB.

All OS/2 multimedia public interfaces, for example error message defines and common definitions, are accessible through the OS2ME.H file. Constants and prototypes for multimedia window control functions, MMIO file services functions, and high-level interfaces are accessible after the following defines are included in your application:

Define	**Services**
#define INCL_SW	Window Control Functions
#define INCL_MMIOOS2	MMIO File Services
#define INCL_MACHDR	High-Level Services

OS/2 multimedia applications should link with the MMPM2.LIB library.

Note: OS/2 multimedia header files have naming conventions compatible with the standard OS/2 format. Applications using previous versions of the MMPM/2 header files will still use those header files by default when the applications are compiled. In order to use the OS/2-consistent header files in an application, define INCL_OS2MM in the program. Defining INCL_OS2MM automatically defines the following:

INCL_MCIOS2	MCI-related include files (MCIOS2.H and MMDRVOS2.H)
INCL_MMIOOS2	MMIO include file (MMIOOS2.H)

All existing applications remain binary compatible. If they are recompiled a choice of which set of headers to use is available. If new header files are used, the source code must be modified to conform to the name changes.

Extendable Device Support

The system architecture of OS/2 multimedia extensions is designed to be extendable. This level of modularity allows independent development of support for new hardware devices, logical media devices, and file formats.

Examples of media control interface devices are listed in the following table. The table shows the logical device types that can be supported and already have media control interface definitions. Devices currently supported by OS/2 multimedia are indicated by (X) marks.

Media Device Type	OS/2 Multimedia	String	Constant
Amplifier mixer	X	ampmix	MCI_DEVTYPE_AUDIO_AMPMIX
Audio tape player		audiotape	MCI_DEVTYPE_AUDIO_TAPE
CD audio player	X	cdaudio	MCI_DEVTYPE_CD_AUDIO
CD-XA player	X	cdxa	MCI_DEVTYPE_CDXA
Digital audio tape		dat	MCI_DEVTYPE_DAT
Digital video player	X	digitalvideo	MCI_DEVTYPE_DIGITAL_VIDEO
Headphone		headphone	MCI_DEVTYPE_HEADPHONE
Microphone		microphone	MCI_DEVTYPE_MICROPHONE
Monitor		monitor	MCI_DEVTYPE_MONITOR
Other		other	MCI_DEVTYPE_OTHER
Video overlay		videooverlay	MCI_DEVTYPE_OVERLAY
Sequencer	X	sequencer	MCI_DEVTYPE_SEQUENCER
Speaker		speaker	MCI_DEVTYPE_SPEAKER
Videodisc player	X	videodisc	MCI_DEVTYPE_VIDEODISC
Video tape/cassette		videotape	MCI_DEVTYPE_VIDEOTAPE
Waveform audio player	X	waveaudio	MCI_DEVTYPE_WAVEFORM_AUDIO

Note: M-Control Program 2.01, which supports the M-Motion Video Adapter/A, provides overlay extensions for OS/2 multimedia.

Media Control Interface

This section describes the services offered to applications by the media control interface for managing devices in the multimedia environment.

Command Message and Command String Interfaces

When a user activates a PM control to use a multimedia device function, the OS/2 multimedia application window procedure sends a command to the media control interface. Depending on the needs of the application, the window procedure can use the command message interface or the command string interface to implement these device commands. Messages for the command message interface (also referred to as procedural interface) are sent with mciSendCommand. Strings for the command string interface are sent to the Media Device Manager for parsing, using the mciSendString function.
See the following figure.

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
|         mciSendCommand                        mciSendString                 |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|              Media Device Manager Interface Layer                           |
M                                                                             |
D                                                                             |
M                                                           Default Tables    |
|                                                                             |
|                                                                 System      |
|                      Table-Driven Parser     `                              |
|                                                                waveaudio    |
|                                                                             |
|                                                                sequencer    |
|                                                                             |
|                                                                 cdaudio     |
|         Media Device Manager                                                |
|                                                                  cdxa       |
|_ _ _ _  _ _ _ _  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _                            |
                                                 |                 ampmix     |
                                                 |                            |
      Media    Media    Media                    |              videodisc     |
      Driver   Driver   Driver                   |                            |
                                                 |             digitalvideo   |
                                                 |                            |
                        Custom                   |                other       |
                        Table                    |                            |
                                                 | _ _ _ _ _ _ _ _ _ _ _ _ _ _|
```

The string interface provides access to most functions of the message interface. However, operations that involve identifying multiple devices (for example, for the purpose of establishing connections), or operations that return complex data structures (such as a CD table of contents) are available only through the message interface.

Each time a message is sent to the Media Device Manager with mciSendCommand, flags are set and a pointer to a data structure is passed. Each time a string is passed with mciSendString, it must be converted to the message format understood by the media driver. The Media Device Manager calls the multimedia string parser, which is case insensitive, to interpret the strings. The time required for this conversion process makes the string method of control slightly slower than the message method. However, the string interface generally requires less application code than the command message interface. The string interface also lets users interactively control devices with a command line or PM interface. See Command Strings.

Command Messages

Command messages are used by the command message interface and specified with mciSendCommand. Most command messages have corresponding string commands that are used by the command string interface and specified with mciSendString. Command messages are sent either to a logical device or to the system. The following table lists the command messages sent to devices. Commands that cause asynchronous responses to be generated, such as cue point and position advise, can be called using the appropriate string command;

however, their responses are returned to window procedures.

Command Messages Supported by All Devices	
MCI_OPEN	Establishes a specific instance of a multimedia device or file.
MCI_GETDEVCAPS	Gets the capabilities of a device.
MCI_INFO	Gets textual information from the device.
MCI_STATUS	Gets the current status of the device.
MCI_CLOSE	Closes the device.
Device Setup Command Messages	
MCI_SET	Changes the configuration of the device.
MCI_CONNECTOR	Enables, disables, or queries the state of a connector.
Playback and Recording Command Messages	
MCI_CUE	Prerolls a device for playing or recording.
MCI_SEEK	Seeks to a specified position in the file.
MCI_PLAY	Begins transmitting output data.
MCI_RECORD	Begins recording data from the specified position.
MCI_PAUSE	Suspends the playing or recording operation.
MCI_RESUME	Resumes the playing or recording operation.
MCI_STOP	Stops the playing or recording operation.
MCI_LOAD	Loads a data element into a media device. An example of a data element is a waveform file.
MCI_SAVE	Saves the current file to disk.
Synchronization Command Messages	
MCI_SET_CUEPOINT	Sets run-time cue points.
MCI_SET_POSITION_ADVISE	Advises the application when time elapses or position changes.
MCI_SET_SYNC_OFFSET	Biases MCI_PLAY starting positions and MCI_SEEK target positions for group operations.
Device-Specific Command Messages	
MCI_CAPTURE	Captures the current video image and stores it as an image device element.
MCI_ESCAPE	Sends a custom message directly to the media driver.

MCI_GETIMAGEBUFFER	Gets the contents of the capture video buffer or the current movie frame.
MCI_GETTOC	Gets a contents structure for the currently loaded CD-ROM disc.
MCI_PUT	Sets the source and destination rectangles for the transformation of the video image. It also sets the size and position of the default video.
MCI_REWIND	Seeks the media to the beginning point.
MCI_SETTUNER	Causes the digital video MCD to change the frequency the tuner device is tuned to.
MCI_SPIN	Spins the videodisc player up or down.
MCI_STEP	Advances or backs up the videodisc player one or more frames.
MCI_WHERE	Returns the source and destination rectangles set by MCI_PUT. It also returns the size and position of the video window.
MCI_WINDOW	Specifies the window in which to display video output, and controls the visibility of the default video window.

Editing Command Messages

MCI_COPY	Copies specified data range into clipboard or buffer.
MCI_CUT	Removes specified data range and places it into clipboard or buffer.
MCI_DELETE	Deletes specified data range. Clipboard or buffer is not used.
MCI_PASTE	Deletes selected data range if difference between FROM and TO is more than zero, then inserts data from buffer or clipboard.
MCI_REDO	Reverses previous MCI_UNDO command.
MCI_UNDO	Cancels previous RECORD, CUT, PASTE, or DELETE.

The following table lists the system command messages specified with mciSendCommand.

MCI_DEVICESETTINGS	Provides a media control interface driver the opportunity to insert custom settings pages.
MCI_GROUP	Makes and breaks device group associations.
MCI_MASTERAUDIO	Sets the system master volume and toggles speakers and headphones.
MCI_SYSINFO	Gets and sets device and system information.
MCI_CONNECTORINFO	Gets information regarding the number and types of connectors

	defined for a device.
MCI_DEFAULT_CONNECTION	Makes, breaks, or queries default connections established for a device.
MCI_CONNECTION	Gets the device context connection or establishes an alias for a connected device.
MCI_ACQUIREDEVICE	Acquires a device for use.
MCI_RELEASEDEVICE	Releases a device from use.

Command Strings

String commands utilize a more English text format than command messages. Following is the valid syntax for passing string commands directly to the media control interface:

```
<COMMAND> <DEVICE_TYPE|DEVICE_NAME|ALIAS|ELEMENT> <PARAMETERS>
```

This format is used for all string commands except **masteraudio**, which does not require a device name. The format for the **masteraudio** command is:

```
<COMMAND> <PARAMETERS>
```

An application calls mciSendString to pass the string command to the Media Device Manager for parsing and execution. The String Test Sample program, provided in the Toolkit (\TOOLKIT\SAMPLES\MM\MCISTRNG), illustrates the interpretive string interface. The following code fragment shows the call to mciSendString in the String Test Sample.

```
ulSendStringRC =
   mciSendString(
    (PSZ) &acMCIString[ 0 ],          /* The MCI String Command     */
    (PSZ) &acMCIReturnString[ 0 ],    /* Place for return strings   */
    (USHORT) MCI_RETURN_STRING_LENGTH, /* Length of return space     */
    hwndDisplayDialogBox,              /* Window to receive notifies */
    usCountOfMCIStringSent );          /* The user parameter         */
```

The following is an example of the string commands required to open a CD player and play an entire CD.

```
open cdaudio01 alias cdaud1 shareable
status cdaud1 media present wait
status cdaud1 mode wait
set cdaud1 time format milliseconds
seek cdaud1 to start
play cdaud1 notify
.
.
.
** play the entire disc **
.
.
.
close cdaud1
```

The **status** commands let the application know if a CD is present and if the drive is ready. Notice that wait flags are used; otherwise the commands would return immediately with no status information. The **set** command sets the time base to milliseconds for all future commands. The **close** command is sent after the application receives an MM_MCINOTIFY message at the completion of the **play** command.

Note: The **close** command can be sent at any time.

Authoring languages that include support for the media control interface can integrate device command strings like these with authoring language syntax to create multimedia presentations. The string interface provides a 16-bit interface to enable developers to integrate multimedia function with the macro languages of existing 16-bit applications.

Wait and Notify Flags

An application can set a wait or a notify flag on a device command sent with mciSendString or mciSendCommand. These two flags are mutually exclusive and are available on all commands except some system commands.

Flag	Description
wait	The command is executed synchronously. The function waits until the requested action is complete before returning to the application.
notify	The command is executed asynchronously, allowing control to be returned immediately to the application. When the requested action is complete, an MM_MCINOTIFY message is sent to the application window procedure.

Note: If a command is issued without a wait flag or notify flag specified, the command is executed asynchronously, and the application is never notified.

The wait flag is useful for operations that are conducted quickly, like the playback of short sounds, which the application wants to complete before it continues. The wait flag is also useful for operations that return information, such as device capabilities, because the Media Device Manager parser converts the return code to a meaningful string. However, the conversion occurs only if the wait flag is specified.

The wait flag should be used with care when issuing commands from threads that read application input message queues as it ties up the thread, preventing all PM messages in the system from being processed while the command issued with the wait flag is executed.

The notify flag is useful for operations that are conducted over a period of time. For example, the playing of a waveform file often can take a while to complete. By specifying the notify flag, an application requests to be notified when processing of the command is complete. The application window procedure can then remain responsive to input queue processing.

OS/2 Multimedia Notification Messages

The system returns notification messages to applications to indicate OS/2 multimedia events such as completing a media device function or passing ownership of a media device from one process to another. Following is a list of OS/2 multimedia notification messages.

Notification Message	Reason for Notification
MM_MCICUEPOINT	A cue point was detected. Cue points are set with the playlist CUEPOINT instruction or the MCI_SET_CUEPOINT command message.
MM_MCIEVENT	A device has generated an event.
MM_MCINOTIFY	A device has completed an action, or an error has occurred.
MM_MCIPASSDEVICE	A shared device is being lost or gained.
MM_MCIPLAYLISTMESSAGE	A MESSAGE instruction was encountered in a playlist.

MM_MCIPOSITIONCHANGE	The time period or position specified with the MCI_SET_POSITION_ADVISE command message has passed.

With the exception of MM_MCIEVENT, the system returns notification messages asynchronously to applications using WinPostMsg; MM_MCIEVENT notifications are returned synchronously with WinSendMsg.

A PM application receives notifications by passing its message queue window handle as a parameter on the mciSendCommand or mciSendString call. Applications can also receive notifications by passing a handle to a Control Program queue. For more information see Using a Control Program Queue for Notifications.

If an application sends a command message with mciSendCommand and specifies the MCI_NOTIFY flag, control returns immediately to the application. The media control interface posts a notification message to the window specified in the callback window handle after the command completes processing. The MM_MCINOTIFY message is returned asynchronously to the application using WinPostMsg. It can have any of the following values:

Notification Code	Meaning
MCI_NOTIFY_SUCCESSFUL	The command completed successfully.
MCI_NOTIFY_SUPERSEDED	Another command is being processed.
MCI_NOTIFY_ABORTED	Another command interrupted this one.

Note: If none of the above notification codes are returned, an error code is returned, indicating that the asynchronous processing of the command ended in an error condition. To convert the error code to a textual description of the error, the application calls the mciGetErrorString function.

The following code fragment illustrates how the Audio Recorder Sample program, provided in the Toolkit (\TOOLKIT\SAMPLES\MM\RECORDER), handles the MM_MCINOTIFY notification message.

```
case MM_MCINOTIFY:
 /*
  * This message is returned to an application when a device
  * successfully completes a command that was issued with a NOTIFY
  * flag, or when an error occurs with the command.
  *
  * This message returns two values. A user parameter (mp1) and
  * the command message (mp2) that was issued. The low word of mp1
  * is the Notification Message Code, which indicates the status of the
  * command like success or failure. The high word of mp2 is the
  * Command Message which indicates the source of the command.
  */

  usNotifyCode = (USHORT) SHORT1FROMMP( mp1);     /* low-word  */
  usCommandMessage = (USHORT) SHORT2FROMMP( mp2); /* high-word */

  switch (usCommandMessage)
  {
   case MCI_PLAY:
      switch (usNotifyCode)
      {
         case MCI_NOTIFY_SUCCESSFUL:
            if (eState != ST_STOPPED)
            {
               /*
                * Update the status line with appropriate message.
                */
               UpdateTheStatusLine(hwnd, IDS_STOPPED);
               eState = ST_STOPPED;

               /*
                * Stop the play button animation
                */
               WinSendMsg( hwndPlayPB,        /* Play button handle */
                           GBM_ANIMATE,       /* Animation control  */
                           MPFROMSHORT(FALSE),/* Animation flag     */
                           NULL );            /* Ignore return data */
            }
            break;

         case MCI_NOTIFY_SUPERSEDED:
         case MCI_NOTIFY_ABORTED:
            /* we don't need to handle these messages. */
            break;

         default:
            /*
             * If the message is none of the above, then it must be
             * a notification error message.
             */
            ShowMCIErrorMessage( usNotifyCode);
            eState = ST_STOPPED;

            /*
             * Stop the play button animation and update the status
             * line with appropriate text.
             */
            WinSendMsg( hwndPlayPB,         /* Play button handle  */
                        GBM_ANIMATE,        /* Animation control   */
                        MPFROMSHORT(FALSE), /* Animation flag      */
                        NULL );             /* Ignore return data  */
            UpdateTheStatusLine(hwnd, IDS_STOPPED);
            break;
      }
      break;
}
return( (MRESULT) 0);
```

Using a Control Program Queue for Notifications

An OS/2 application that does not have a PM window, and therefore cannot use a PM message queue for receiving notification messages, can use an OS/2 Control Program queue instead. The Control Program queue method of notification should also be considered for time-critical PM applications, because it is faster than using the PM message queue.

To receive notifications on a Control Program queue, the application must specify the MCI_DOS_QUEUE flag with the MCI_OPEN command message when using mciSendCommand (or the **dosqueue** keyword with the **open** command when using mciSendString). This flag indicates that the window handle specified for receiving notification messages is actually a handle to a Control Program queue, not a PM window.

The application issues DosReadQueue, which mimics the PM message queue function by blocking until there is something in the queue to process. To place a notification message in the queue, the system issues DosWriteQueue and specifies the queue handle given to it by the application.

The syntax for Control Program queues and PM message queues varies slightly. A typical PM message queue process function has the following four parameters:

```
HWND         hwnd   /* Window handle       */
ULONG        msg    /* Notification type   */
MPARAM       mp1    /* Message parameter 1 */
MPARAM       mp2    /* Message parameter 2 */
```

The DosReadQueue call has eight parameters. The system uses the first four parameters of DosReadQueue in the same manner as a PM message queue process function.

```
HQUEUE       hq     /* Queue handle */
PREQUESTDATA pRD    /* Pointer to REQUESTDATA */
PULONG       pmp1   /* Pointer to mp1 information */
PPVOID       pmp2   /* Pointer to mp2 information */
```

The second parameter of DosReadQueue points to the REQUESTDATA structure, which contains a *ulData* field. This field corresponds to the *msg* field of PM message queues. The application can use the remaining parameters of DosReadQueue for its own purposes.

The following example code illustrates how an application can use Control Program queue functions to handle multimedia notification messages.

```
{
 PID      OwnerPID=0;
 HQUEUE   QHandle =0;
 char     ch, retstring[100], QueueName[100];

 strcpy(QueueName, "\\QUEUES\\");
 if (argc == 2) strcat(QueueName, argv[1]);
 else           strcat(QueueName, "DEFAULT");
 if (DosCreateQueue(&QHandle, QUE_FIFO, QueueName)) return(1L);
 if (DosOpenQueue(&OwnerPID, &QHandle, QueueName))  return(1L);
 _beginthread( QMonitor, (PVOID)NULL, 65536L, (PVOID)QHandle);
 SendString((LPSTR)"open d:\\mmos2\\sounds\\applause.wav shareable
             dosqueue wait alias a",
            (LPSTR)retstring, 100,
            (HWND)QHandle, 0);
 SendString((LPSTR)"setpositionadvise a on every 3000 wait",
            (LPSTR)retstring, 100,
            (HWND)QHandle, 0);
 SendString((LPSTR)"play a notify", (LPSTR)retstring, 100, (HWND)QHandle, 0);
 ch=(char)getch();
 SendString((LPSTR)"close a wait", (LPSTR)retstring, 100, (HWND)QHandle, 0);
 DosCloseQueue(QHandle);
 return(0L);
}

VOID _Optlink QMonitor( PVOID qh)
{
 APIRET      rc=0;
 BYTE        Priority;
 REQUESTDATA RD;
 ULONG       msg, mp1, mp2;

 printf("QMonitor Started!\n");
 while (1)
  {
   DosReadQueue((HQUEUE)qh, &RD, (PULONG)&mp1, (PPVOID)&mp2, 0L, (BOOL32)0,
                &Priority, (HEV)0);
   switch(RD.ulData)
     {
       case MM_MCINOTIFY:
          printf("   msg=MM_MCINOTIFY:\n");
          printf("   mp1=%d\n",mp1);
          printf("   mp2=%d\n",mp2);
          break;
       case MM_MCIPASSDEVICE:
          printf("   msg=MM_MCIPASSDEVICE:\n");
          printf("   Device id=%d\n",mp1);
          switch (mp2)
            {
              case MCI_LOSING_USE:
                 printf("   MCI_LOSING_USE:\n");
                 break;
              case MCI_GAINING_USE:
                 printf("   MCI_GAINING_USE:\n");
                 break;
              default:
                 printf("   mp2=%d\n",mp2);
                 break;
            }
          break;
       case MM_MCIPOSITIONCHANGE:
          printf("   msg=MM_MCIPOSITIONCHANGE:\n");
          printf("   mp1=%d\n",mp1);
          printf("   mp2=%d\n",mp2);
          break;
       case MM_MCICUEPOINT:
          printf("   msg=MM_MCICUEPOINT:\n");
          printf("   mp1=%d\n",mp1);
          printf("   mp2=%d\n",mp2);
          break;
       case MM_MCIPLAYLISTMESSAGE:
          printf("   msg=MM_MCIPLAYLISTMESSAGE:\n");
          printf("   mp1=%d\n",mp1);
          printf("   mp2=%d\n",mp2);
          break;
       case MM_MCIEVENT:
          printf("   msg=MM_MCIEVENT:\n");
          printf("   mp1=%d\n",mp1);
          printf("   mp2=%d\n",mp2);
          break;
       default:
          printf("   msg=%d\n",RD.ulData);
          printf("   mp1=%d\n",mp1);
          printf("   mp2=%d\n",mp2);
          break;
     }
  }   /* endwhile */
 return;
}


VOID SendString( LPSTR string, LPSTR retstring, ULONG retsize, HWND Handle,
                 ULONG userparm)
{
 ULONG rc;
 rc=mciSendString(string, retstring, retsize, Handle, userparm);
 if (rc) printf("Error: (%s) rc=%d\n",string,rc);
 return;
}
/* QUEUE.H */
#define INCL_BASE
#define INCL_32

#include<os2.h>
#include<os2me.h>
#include<mmioos2.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

int           main(int argc, char *argv[], char *envp[]);
VOID          SendString(LPSTR string,
                         LPSTR retstring,
                         ULONG retsize,
                         HWND Handle,
                         ULONG userparm);
VOID _Optlink QMonitor( PVOID qh);
```

Time Formats for Device Commands

Media position and time information are required as input and also returned as output by many multimedia commands. Time formats vary, depending on the device being used and the format of the data being operated on. The default time base for both the procedural and string interfaces is MMTIME. See MMTIME Format.

Other time formats, such as milliseconds, are also supported.

Time formats used by media control interface devices for measuring time are listed in the following table. The flags shown in the table are set with the MCI_SET command.

Device	Formats	Flags
CD-DA	milliseconds	MCI_FORMAT_MILLISECONDS
	mmtime	MCI_FORMAT_MMTIME
	minutes-seconds-frames	MCI_FORMAT_MSF
	tracks-min-sec-frame	MCI_FORMAT_TMSF
CD-XA	milliseconds	MCI_FORMAT_MILLISECONDS
	mmtime	MCI_FORMAT_MMTIME
digital video	milliseconds	MCI_FORMAT_MILLISECONDS

	mmtime	MCI_FORMAT_MMTIME
	frames	MCI_FORMAT_FRAMES
	hours-minutes-seconds	MCI_FORMAT_HMS
	hours-min-sec-frames	MCI_FORMAT_HMSF
waveform audio	milliseconds	MCI_FORMAT_MILLISECONDS
	mmtime	MCI_FORMAT_MMTIME
	bytes	MCI_FORMAT_BYTES
	samples	MCI_FORMAT_SAMPLES
MIDI sequencer	milliseconds	MCI_FORMAT_MILLISECONDS
	mmtime	MCI_FORMAT_MMTIME
	SMPTE 24	MCI_SEQ_SET_SMPTE_24
	SMPTE 25	MCI_SEQ_SET_SMPTE_25
	SMPTE 30	MCI_SEQ_SET_SMPTE_30
	SMPTE 30	MCI_SEQ_SET_SMPTE_30DROP
	song pointer	MCI_SEQ_SET_SONGPTR

MMTIME Format

MMTIME is a standard time and media position format supported by the media control interface. This time unit is 1/3000 second, or 333 microseconds. Conversion macros are provided for convenient conversion of other popular time formats to and from this format. MMTIME values are passed as long (32-bit) integer values.

To use MMTIME on command messages, send the MCI_SET message specifying the MCI_SET_TIME_FORMAT flag. Use MCI_FORMAT_MMTIME in the *ulTimeFormat* field of the MCI_SET_PARMS structure.

The macros shown in the following figure are available for conversion to and from the MMTIME format.

Conversion to MMTIME	Conversion to Other Formats
REDBOOKTOMM (ULONG)	REDBOOKFROMMM (ULONG)
FPS24TOMM (ULONG)	FPS24FROMMM (ULONG)
FPS25TOMM (ULONG)	FPS25FROMMM (ULONG)
FPS30TOMM (ULONG)	FPS30FROMMM (ULONG)
MSECTOMM (ULONG)	MSECFROMMM (ULONG)
HMSTOMM (ULONG)	HMSFROMMM (ULONG)

Packed Time Formats

The packed time formats described in the following sections require that the application format the ULONG value passed in command message parameter structures. When these values are passed in string commands, any value containing a colon (:) is assumed to be a field-oriented value. For example, if the time format for a CD audio device is set to TMSF, and the value 4:10:00:00 is specified, this value is interpreted as track 4, 10 minutes, 0 seconds, and 0 frames. However, if the value 4100000 is specified, the integer is passed directly, and the assignment to byte fields is quite different.

It is not required that a field-oriented value contain specifications for all fields. For example, the following are equivalent specifications for track 4:

```
4:00:00:00
4:00:00
4:00:
4:00
4:
4
```

The interpretation of field-oriented values is left-justified with respect to the placement of colons. Values not specified default to zero. If a value has a colon, it is subject to field-oriented interpretation, regardless of the time format currently set for the device.

HMSF (SMPTE) Packed Time Format

The HMSF packed time format represents elapsed hours, minutes, seconds, and frames from any specified point. This time format is packed into a 32-bit ULONG value as follows:

```
 High-Order Byte Low-Order Byte  High-Order Byte Low-Order Byte

 Frames          Seconds         Minutes         Hours
```

MSF Packed Time Format

The CD-DA MSF time format, also referred to as the Red Book time format, is based on the 75-frame-per-second CD digital audio standard. Media position values in this format are packed into a 32-bit ULONG value as follows:

```
 High-Order Byte Low-Order Byte  High-Order Byte Low-Order Byte

 Reserved        Frames          Seconds         Minutes
```

The following macros aid in extracting information in packed MSF format:

Macro	**Description**
MSF_MINUTE(time)	Gets the number of minutes.
MSF_SECOND(time)	Gets the number of seconds.
MSF_FRAME(time)	Gets the number of frames.

For example, the following code fragment sets the time in *ulTime* to 6 minutes and 30 seconds (06:30:00).

```
ULONG ulTime;
.
.
.
MSF_MINUTE(ulTime) = 6
MSF_SECOND(ulTime) = 30;
MSF_FRAME(ulTime) = 0;
```

TMSF Packed Time Format

The CD-DA TMSF time format is based on the 75-frame-per-second CD digital audio standard. Media position values in this format are packed into a 32-bit ULONG value as follows:

```
 High-Order Byte Low-Order Byte  High-Order Byte Low-Order Byte

 Frames          Seconds         Minutes         Track
```

The following macros aid in extracting information in packed TMSF format:

Macro	**Description**
TMSF_TRACK(time)	Gets the number of tracks.
TMSF_MINUTE(time)	Gets the number of minutes.
TMSF_SECOND(time)	Gets the number of seconds.
TMSF_FRAME(time)	Gets the number of frames.

For example, the following code fragment sets the time in *ulTime* to 2 minutes into track 2 (02:02:00:00).

```
ULONG ulTime;
.
.
.
TMSF_TRACK(ulTime) = 2;
TMSF_MINUTE(ulTime) = 2;
TMSF_SECOND(ulTime) = 0;
TMSF_FRAME(ulTime) = 0;
```

Note: MSF and TMSF macros can be found in the MCIOS2.H file.

HMS Packed Time Format

The HMS packed time format, representing hours, minutes, and seconds, is packed into a 32-bit ULONG value as follows:

High-Order Byte	Low-Order Byte	High-Order Byte	Low-Order Byte
Reserved	Seconds	Minutes	Hours

Opening a Media Device

Media devices are categorized as simple or compound devices. A compound device is an internal device that operates on data objects, such as files, within the system. These data objects are referred to as device elements. A simple device is an external device that does not require a device element.

Media device types supported by OS/2 multimedia are shown in the following table.

Logical Device Type	String	Constant
Amplifier-mixer	ampmix	MCI_DEVTYPE_AUDIO_AMPMIX
CD-DA player	cdaudio	MCI_DEVTYPE_CD_AUDIO
CD-XA player	cdxa	MCI_DEVTYPE_CDXA_PLAYER
Digital video player	digitalvideo	MCI_DEVTYPE_DIGITAL_VIDEO
MIDI sequencer	sequencer	MCI_DEVTYPE_SEQUENCER
Waveform audio player	waveaudio	MCI_DEVTYPE_WAVEFORM_AUDIO
Videodisc player	videodisc	MCI_DEVTYPE_VIDEODISC

Device type constants represent one way of specifying devices in command messages. String names can be specified in either command messages or command strings.

To use the string interface to communicate with a device, an application calls mciSendString and passes the textual command **open**. Following is the syntax used for the textual command:

open *device_name* <shareable> <type *device_type* > <alias *alias* >

Parameters for the open command are:

Parameters	Description
device_name	Specifies the name of a device or device element.
shareable	Indicates the device or device element may be shared by other applications.
type *device_type*	Specifies the device type when *device_name* is a device element.
alias *alias*	Specifies an alternate name for the device.

Here is an example of the syntax for opening a device:

```
open horns.wav type waveaudio alias sound1
```

where "horns.wav" is the device element and "waveaudio" is the device type.

The system also supports a shortcut version of the syntax:
open *device_type*!*element_name*

The shortcut version of the previous example looks like this:

```
open waveaudio!horns.wav alias sound1
```

File Type Associations

A specific device can have file extensions and .TYPE EAs (Extended Attributes) associated with it. The OS/2 multimedia user can map a file extension or .TYPE EA to a specific device with the Multimedia Setup application located in the Multimedia folder. An OS/2 multimedia subsystem developer writing an installation DLL can map a file extension or .TYPE EA to a device using MCI_SYSINFO_SET_EXTENSIONS or MCI_SYSINFO_SET_TYPES. For an extension or .TYPE EA to be mapped to a device, it must be unique across installation names.

For example, the Multimedia Setup application can be used to associate the WAV extension with the waveaudio01 device. The device can then be opened by passing the name of a data element with a WAV extension as a parameter in the open command to mciSendString. Suppose the following string is passed:

```
open honk.wav wait
```

The waveaudio01 device is opened with the data file honk.wav.

Default and Specific Devices

The following table shows some examples of open commands. A default device is opened if only a logical device type (for example, waveaudio) is specified in the open command. The default device for a logical device type can be queried and set by the user with the Multimedia Setup application. The default device also can be queried and set with MCI_SYSINFO by an installation DLL for a media device.

A specific device is opened by specifying its name (for example waveaudio01), or by specifying a device element with an extension or .TYPE EA that is associated with the device.

Open Command	Description
open waveaudio	Opens a default device of type waveaudio.
open waveaudio01	Opens a specific device of type waveaudio.
open foo.xyz	Opens a specific device that is associated with the .TYPE EA (if any) of foo.xyz; otherwise opens a specific device that has a unique association with file extension xyz; otherwise returns MCIERR_INVALID_DEVICE_NAME.

Shareable Flag

By setting the shareable flag for an open request, an application can share an OS/2 multimedia device with other applications. To enable device sharing, the multimedia system posts the MM_MCIPASSDEVICE message with WinPostMsg to the application. The message informs the application the device context is becoming active (MCI_GAINING_USE) or inactive (MCI_LOSING_USE).

After the application processes the MCI_GAINING_USE event notification, it can issue device commands. The device context becomes inactive when the MCI_LOSING_USE event notification is received.

If the application has specified a notify flag on the open, the receipt of an MM_MCINOTIFY message does not mean the device context is active. When MCI_NOTIFY_SUCCESSFUL is received, the commands **status**, **capability**, and **info** can be issued, because the multimedia system allows these commands to be made to inactive instances. If the application issues a command to an inactive instance and the instance must be active to process the command, the multimedia system returns MCIERR_INSTANCE_INACTIVE.

When an application opens a device without setting the shareable flag, the Media Device Manager attempts to acquire the device for the exclusive use of the application. If a device context already exists that was either opened as nonshareable by another application or opened as shareable but then acquired exclusively by another application, the open fails and the application receives the MCIERR_DEVICE_LOCKED error code. The application can subsequently make the device context shareable by issuing an MCI_RELEASEDEVICE message.

See Device Sharing By Applications for more detailed information on device sharing.

Device Alias

When a device is opened, it can be given an alias, or alternate name. The primary use of a device alias is to simplify the specifying of subsequent commands to control the device through the string interface. A device alias is referenced only from the string interface, and it is valid only within the process that opened the device context.

For example, the following strings can be passed with mciSendString:

```
open horns.wav alias honk
play honk
```

A secondary use of the device alias is to differentiate between device contexts opened by the same process. For example:

```
open horns.wav alias honk
open bells.wav alias ring
play ring wait
play honk wait
```

Note: The maximum length for an alias is 20 characters. Placing an alias in quotation marks is permitted.

When a device is opened using the string interface, a device context ID is returned. If the application provides a return buffer in the call to mciSendString, the ID can be used to issue commands to the device context using the mciSendCommand interface, when necessary.

Using the Command Message Interface

To use the command message interface to communicate with a device, an application calls mciSendCommand and passes the command message MCI_OPEN. If the request is successful, a device handle for access to the device context is returned in the *usDeviceID* field of the MCI_OPEN_PARMS data structure. This handle is retained for use in subsequent message commands.

An alias can be specified with the MCI_OPEN_ALIAS flag in the command message MCI_OPEN. Commands can then be issued to the device context by means of the string interface.

The following code fragment shows the opening of devices in the Duet Player I sample program. The *hwndCallback* field contains the application's main window procedure so that the MM_MCIPASSDEVICE messages are sent to it when the duet player gains or passes control of the device. The device ID and type fields of the structure are not needed because the audio file name is specified as the element field of the structure. This causes the Media Device Manager (MDM) to open the appropriate device based on the file name extension. Once the MCI_OPEN_PARMS structure is initialized, an MCI_OPEN command is specified with the mciSendCommand function for each separate part of the duet.

```
 /*
  * Open one part of the duet. The first step is to initialize an
  * MCI_OPEN_PARMS data structure with the appropriate information,
  * then issue the MCI_OPEN command with the mciSendCommand function.
  * We will be using an open with only the element name specified.
  * This will cause the default connection, as specified in the
  * MMPM.INI file, for the data type.
  */
mopDuetPart.hwndCallback   = (ULONG)  hwnd; /* For MM_MCIPASSDEVICE */
mopDuetPart.usDeviceID     = (USHORT) NULL; /* this is returned    */
mopDuetPart.pszDeviceType  = (PSZ)    NULL; /* using default conn. */
mopDuetPart.pszElementName = (PSZ)   aDuet[sDuet].achPart1;


     ulError = mciSendCommand( (USHORT) 0,
                               MCI_OPEN,
                               MCI_WAIT | MCI_OPEN_ELEMENT |
                               MCI_OPEN_SHAREABLE | MCI_READONLY,
                               (PVOID) &mopDuetPart,
                               UP_OPEN);


     if (!ulError)  /* if we opened part 1 */
     {
        usDuetPart1ID = mopDuetPart.usDeviceID;

        /*
         * Now, open the other part
         */
        mopDuetPart.pszElementName    = (PSZ)   aDuet[sDuet]achPart2;

        ulError = mciSendCommand( (USHORT) 0,
                                  MCI_OPEN,
                                  MCI_WAIT | MCI_OPEN_ELEMENT |
                                  MCI_OPEN_SHAREABLE | MCI_READONLY,
                                  (PVOID) &mopDuetPart,
                                  UP_OPEN);

        if (!ulError)  /* if we opened part 2 */
        {
           usDuetPart2ID = mopDuetPart.usDeviceID;
```

Memory Playlists

In addition to specifying files or Resource Interchange File Format (RIFF) chunks to be loaded by compound devices, you also can specify memory objects. You create memory objects, for example, to play synthesized audio using the waveform audio media driver. These memory objects can be placed under the control of the memory playlist.

The memory playlist is a data structure in an application. It contains an array of simple, machine-like instructions you formulate, each of which has a fixed format consisting of a 32-bit operation code and three 32-bit operands. Playlist instructions are described in the following table.

To have playlist instructions interpreted by the playlist processor, you specify the MCI_OPEN_PLAYLIST flag with the MCI_OPEN command message. This flag indicates that the *pszElementName* field in the MCI_OPEN_PARMS data structure is a pointer to a memory playlist.

Using playlist instructions, you can play audio objects in succession from one or more memory buffers. Instructions include branching to and returning from subroutines within the playlist. In addition, the playlist can be modified dynamically by the application while it is being played. Because less overhead is involved when playing audio data from memory, playlist programs will have higher performance. If your application requires speed or if it needs to modify the data before it is sent to the audio device, use playlists.

Command	Description
BRANCH_OPERATION	Transfers control to another instruction in the playlist. Operand 1-Ignored. Operand 2-The absolute instruction number in the playlist to which control is being transferred. Because the playlist is defined as an array of structures (instruction, operation, and operand values) its first instruction is referenced as array element, index 0. Therefore, the first instruction in the list is 0, the second instruction is 1, and so on. Operand 3-Ignored. Branching out of a subroutine is not prohibited; however, it is not recommended because an unused return address is left on the stack maintained by the playlist processor. An application can enable or disable a BRANCH_OPERATION by exchanging it with a NOP_OPERATION. Operands for a NOP_OPERATION are ignored.
CALL_OPERATION	Transfers control to the absolute instruction number specified in Operand 2, saving the number of the instruction following the CALL for use on a RETURN instruction. CALL instructions may be nested up to 20 levels. Operand 1-Ignored. Operand 2-Absolute instruction number in the playlist to which control is being transferred. Operand 3-Ignored.
CUEPOINT_OPERATION	Causes a cue point data record to be entered into the data stream. Note that the cue point is relative to the DATA_OPERATION that follows it. Operand 1-User-defined parameter to be returned as the low word of *ulMsgParam1* in the MM_MCICUEPOINT message. Operand 2-Offset in MMTIME units for the actual time the CUEPOINT message should be generated. Operand 3-Ignored. The MM_MCICUEPOINT message is returned to the application as soon as possible after the cue point data record is encountered in the data stream. The message is sent to the window handle specified when the device was originally opened. Note: The CUEPOINT instruction is ignored when used in a recording operation.
DATA_OPERATION	Specifies a data buffer to be played from or recorded into. Operand 1-Long pointer to a buffer in the application. Operand 2-Length of the buffer pointed to by Operand 1. Operand 3-Current position in the buffer. This operand is updated by the system during a recording or playback operation. For a playback operation, it is the number of bytes that have been sent to the output device handler. For a recording operation, it is the number of bytes that have been placed into a user buffer. The current position in the buffer is particularly important after a recording operation, because this field contains the number of bytes of recorded data. The remaining bytes in the buffer are not valid. This field is initialized to zero when the DATA_OPERATION statement is first

encountered.
The buffer indicated by the DATA instruction must only contain the raw data bytes from the device and cannot include any header information. Therefore, the precise meaning or format of the data is dependent on the current settings of the media device. For example, a wave audio data element is assumed to have the format PCM or ADPCM, number of bits per sample, and so on, that is indicated by the settings of the audio device.

EXIT_OPERATION

Indicates the end of the playlist.
Operand 1-Ignored.
Operand 2-Ignored.
Operand 3-Ignored.

LOOP_OPERATION

Controls iteration in a playlist. It is the responsibility of the application to initialize the current iteration. The current iteration is reset to zero following loop termination.
Operand 1-Number of times the loop is to be executed.
Operand 2-Target instruction to branch to, when the loop condition fails.
Operand 3-Current iteration.
The last instruction in a loop is a branch back to the LOOP_OPERATION. The operation of the LOOP_OPERATION instruction is as follows:
1. If Operand 3 is less than Operand 1, control is transferred to the playlist instruction following the LOOP instruction, and the iteration count in Operand 3 is incremented.
2. Otherwise, the iteration count is reset to zero and control is passed to the instruction specified in Operand 2.
Typically, the application sets the iteration count to zero when the playlist is passed to the device, but this is not required. The loop instruction merely compares the loop count with the iteration count. If the iteration count is set to a value other than zero when the playlist is passed in, it is as if the loop has been executed that number of times. Also, if a playback operation is stopped, and then the same playlist is loaded again, the loop iteration count is not initialized by the playlist processor.
It is the application's responsibility to see that iteration count values are what is required when switching from play to record, record to play, and when changing settings for the data (for example, *bitspersample*, *samplespersec*, and so on) with the set command. These commands cause the playlist stream to be destroyed and re-created, and the playlist to be reassociated as a new playlist with the playlist processor.

MESSAGE_OPERATION

Returns a message to the application during playlist processing.
Operand 1-Ignored.
Operand 2-ULONG that is returned to the application in the MM_MCIPLAYLISTMESSAGE message MsgParam2.
Operand 3-Ignored.
Each time the playlist processor encounters a MESSAGE instruction, MM_MCIPLAYLISTMESSAGE is returned to the application. MESSAGE instructions can be used by the application to trace specific points during the execution of the playlist processor. The message is sent to the window handle specified when the device was originally opened.
This function is not intended to be used for timing of data production or consumption identified by previously interpreted instructions. Do not rely on the MESSAGE instruction to indicate precisely when a particular piece of digital audio has been played by an audio device; however, the MESSAGE instruction can be used to indicate when a buffer has been consumed and needs to be refilled.

NOP_OPERATION

Used as a placeholder.
Operand 1-Ignored.
Operand 2-Ignored.
Operand 3-Ignored.

RETURN_OPERATION

Transfers control to the playlist instruction following the most recently executed CALL instruction.
Operand 1-Ignored.
Operand 2-Ignored.
Operand 3-Ignored.

SEMPOST_OPERATION

Causes the playlist processor to post an event semaphore. The playlist processor will call DosWaitEventSem.
Operand 1-Contains the semaphore to post.
Operand 2-Ignored.
Operand 3-Ignored.

SEMWAIT_OPERATION

Causes the playlist processor to wait on a semaphore. The playlist processor will call DosWaitEventSem.
Operand 1-Contains the semaphore to perform the wait on.
Operand 2-Amount of time the semaphore should wait.
Operand 3-Boolean value indicating whether or not the semaphore should be cleared before waiting.

Clock Sample Program Playlist Example

The data structure in the following figure holds the playlist that is used to play the chimes in the Clock Sample program provided in the Toolkit (\TOOLKIT\SAMPLES\MM\CLOCK). Note that the definitions for the playlist operation codes can be found in the MCIOS2.H file.

```
/*
 * This double array holds the playlists that will be used to play the
 * chimes for the clock.  Each array has three fields within the
 * structure: one for the playlist command (32-bit value) and three
 * operands (32-bit values).  The DATA_OPERATION's first operand will
 * contain the address to the respective waveform buffers.  Once the
 * playlist has been played, the CHIME_PLAYING_HAS_STOPPED message
 * will be sent so that the application knows that the audio has
 * finished.
 * The clock will have a unique chime for each quarter hour.
 * There are three chime files that are used in different combinations
 * to create all of the chimes used for the clock.  These three files
 * are CLOCK1.WAV, CLOCK2.WAV, and CLOCK3.WAV.
 *
 * The first playlist will play quarter hour chime.  This is simply
 * CLOCK1.WAV.
 *
 * The second playlist will play the half hour chime.  This
 * consists of CLOCK1.WAV + CLOCK2.WAV.
 *
 * The third playlist will play the three quarter hour chime.  This
 * consists of CLOCK1.WAV + CLOCK2.WAV + CLOCK1.WAV.
 *
 * The fourth playlist plays the hour chime.  This consists of
 * CLOCK1.WAV + CLOCK2.WAV + CLOCK1.WAV + CLOCK2.WAV +
 * (HOUR * CLOCK3.WAV)
 * The Number of loops to perform for the hour value is kept in
 * the first operand.  This will be set in a later procedure when the
 * hour of the chime time is known.
 */
PLAY_LIST_STRUCTURE_T apltPlayList[ NUMBER_OF_PLAYLISTS ]
                                  [ NUMBER_OF_COMMANDS ] =
{
   /*
    * Quarter Hour Chime.
    */
   {
      DATA_OPERATION,    0, 0, 0,      /* Chime file 1.  */
      MESSAGE_OPERATION, 0, CHIME_PLAYING_HAS_STOPPED, 0,
      EXIT_OPERATION,    0, 0, 0
   },
   /*
    * Half Hour Chime.
    */
   {
      DATA_OPERATION,    0, 0, 0,      /* Chime file 1.  */
      DATA_OPERATION,    0, 0, 0,      /* Chime file 2.  */
      MESSAGE_OPERATION, 0, CHIME_PLAYING_HAS_STOPPED, 0,
      EXIT_OPERATION,    0, 0, 0
   },
   /*
    * Three Quarter Hour Chime.
    */
   {
      DATA_OPERATION,    0, 0, 0,      /* Chime file 1.  */
      DATA_OPERATION,    0, 0, 0,      /* Chime file 2.  */
      DATA_OPERATION,    0, 0, 0,      /* Chime file 1.  */
      MESSAGE_OPERATION, 0, CHIME_PLAYING_HAS_STOPPED, 0,
      EXIT_OPERATION,    0, 0, 0
   },
 /*
  * Hour Chime.
  */
 {
    DATA_OPERATION,    0, 0, 0, /* Chime file 1.          < Line 0 >*/
    DATA_OPERATION,    0, 0, 0, /* Chime file 2.          < Line 1 >*/
    DATA_OPERATION,    0, 0, 0, /* Chime file 1.          < Line 2 >*/
    DATA_OPERATION,    0, 0, 0, /* Chime file 2.          < Line 3 >*/
    DATA_OPERATION,    0, 0, 0, /* Chime file 3.          < Line 4 >*/
    LOOP_OPERATION,    0, 4, 0, /* Which line to loop on. < Line 5 >*/
    MESSAGE_OPERATION, 0, CHIME_PLAYING_HAS_STOPPED, 0,
    EXIT_OPERATION,    0, 0, 0
 }
```

To prevent lost data, the address range of memory buffers used in DATA operations should not overlap.

Setting up the Playlist

Playlists operate on data from memory. Therefore, space must be allocated for the memory that will be utilized with the playlist.

The SetupPlaylist procedure is performed once, during initialization of the Clock Sample program. It calls the procedure CopyWaveformIntoMemory to copy the waveform files into memory buffers. It also initializes the playlist data structure by supplying the address and size of the memory buffers holding the data in the appropriate data structure fields.

```
VOID SetupPlayList( VOID )
{
 /*
  * This array keeps the address of each audio chime file.
  */
 static LONG *pulBaseAddress[ NUMBER_OF_CHIME_FILES ];

 USHORT usChimeFileId;            /* Chime audio file ID.        */
 ULONG  ulSizeOfFile,             /* Size of audio file.         */

ulMemoryAllocationFlags = PAG_COMMIT | PAG_READ | PAG_WRITE;
for(usChimeFileId=0; usChimeFileId<NUMBER_OF_CHIME_FILES;
    usChimeFileId++)
 {

    ulSizeOfFile = HowBigIsTheChimeFile( usChimeFileId );
    /*
     * If the returned file size is zero, there is a problem with the
     * chime files.  A message will already have been shown to the
     * user by the HowBigIsTheChimeFile function so get out of
     * this routine.
     */
    if ( ulSizeOfFile == 0 )
    {
       return;
    }
    if ( (pulBaseAddress[ usChimeFileId ] = (LONG *)
            malloc( ulSizeOfFile )) == (LONG *) NULL )
    {
    /*
     * The memory for the waveform files cannot be allocated.
     * Notify the user and return from this routine.  No playlist can
     * be created/played until memory is available.
     */
    ShowAMessage(acStringBuffer[IDS_NORMAL_ERROR_MESSAGE_BOX_TEXT - 1 ],
                 IDS_CANNOT_GET_MEMORY, /* ID of the message to show. */
                 MB_OK | MB_INFORMATION |
                 MB_HELP |  MB_APPLMODAL |
                 MB_MOVEABLE );      /* Style of the message box.  */

       return;

    }  /* End of IF allocation fails. */
    /*
     * Place the waveform files into the memory buffer that was just
     * created.
     */
    CopyWaveformIntoMemory(
       pulBaseAddress[ usChimeFileId ],
       ulSizeOfFile,
       usChimeFileId );
    /*
     * Now that we've loaded the waveform into memory, we need to put
     * its address and size into the playlist data statements that
     * use this particular file.
     *
     * Its address must be placed into the data statement's first
     * operand and its size must be placed in the data
     * statement's second operand.
     *
     * For the four different playlists, one for each chime time
     * (1/4, 1/2, 3/4 and 1 hour increments),
     * the address of the chime file and its size will be loaded
     * into each data statement of the Playlist.
     */
    if ( usChimeFileId == 0 )
    /* If we just loaded CLOCK1.WAV */
    {
    /*
     * Put the address of this chime into the first operand of
     * every data operation that uses this particular chime.
     */
    apltPlayList[ 0 ][ 0 ].ulOperandOne =  /* 1/4 hour 1st data op */
    apltPlayList[ 1 ][ 0 ].ulOperandOne =  /* 1/2 hour 1st data op */
    apltPlayList[ 2 ][ 0 ].ulOperandOne =  /* 3/4 hour 1st data op */
    apltPlayList[ 2 ][ 2 ].ulOperandOne =  /* 3/4 hour 3rd data op */
    apltPlayList[ 3 ][ 0 ].ulOperandOne =  /* 1   hour 1st data op */
    apltPlayList[ 3 ][ 2 ].ulOperandOne =  /* 1   hour 3rd data op */
       (ULONG) pulBaseAddress[ usChimeFileId ];  /*    address     */
    /*
     * Now put the size of the file into the second operand of every
     * data operation that uses this particular chime.
     */
    apltPlayList[ 0 ][ 0 ].ulOperandTwo =  /* 1/4 hour 1st data op */
    apltPlayList[ 1 ][ 0 ].ulOperandTwo =  /* 1/2 hour 1st data op */
    apltPlayList[ 2 ][ 0 ].ulOperandTwo =  /* 3/4 hour 1st data op */
    apltPlayList[ 2 ][ 2 ].ulOperandTwo =  /* 3/4 hour 3rd data op */
    apltPlayList[ 3 ][ 0 ].ulOperandTwo =  /* 1   hour 1st data op */
    apltPlayList[ 3 ][ 2 ].ulOperandTwo =  /* 1   hour 3rd data op */
       ulSizeOfFile;                             /* size         */
    }
    else
    if ( usChimeFileId == 1 )
    /* If we just loaded CLOCK2.WAV */
    {
    /*
     * Put the address of this chime into the first operand of
     * every data operation that uses this particular chime.
```

```
     */
    apltPlayList[ 1 ][ 1 ].ulOperandOne =  /* 1/2 hour 2nd data op */
    apltPlayList[ 2 ][ 1 ].ulOperandOne =  /* 3/4 hour 2nd data op */
    apltPlayList[ 3 ][ 1 ].ulOperandOne =  /* 1   hour 2nd data op */
    apltPlayList[ 3 ][ 3 ].ulOperandOne =  /* 1   hour 4th data op */
       (ULONG) pulBaseAddress[ usChimeFileId ];  /* address    */
    /*
     * Now put the size of the file into the second operand of every
     * data operation that uses this particular chime.
     */
    apltPlayList[ 1 ][ 1 ].ulOperandTwo =  /* 1/2 hour 2nd data op */
    apltPlayList[ 2 ][ 1 ].ulOperandTwo =  /* 3/4 hour 2nd data op */
    apltPlayList[ 3 ][ 1 ].ulOperandTwo =  /* 1   hour 2nd data op */
    apltPlayList[ 3 ][ 3 ].ulOperandTwo =  /* 1   hour 4th data op */
       ulSizeOfFile;                             /* size        */
 }
 else
 if ( usChimeFileId == 2 )
 /* If we just loaded CLOCK3.WAV ,       */
 /* (this is the gong part of the chime) */
{
    /*
     * Put the address of this chime into the first operand of
     * every data operation that uses this particular chime.
     */
    apltPlayList[ 3 ][ 5 ].ulOperandOne =  /* 1 hour 5th data op */
       (ULONG) pulBaseAddress[ usChimeFileId ];

    /*
     * Now put the size of the file into the second operand of every
     * data operation that uses this particular chime.
     */

    apltPlayList[ 3 ][ 5 ].ulOperandTwo =  /* 1 hour 5th data op */
       ulSizeOfFile;
  }

 }  /* End of For loop of chime files. */

}  /* End of SetupPlayList */
```

Dynamic Playlist Modification

An application can change a playlist dynamically to achieve various effects in controlling the data stream in memory. In the case of the Clock Sample program, if the chime is an hour chime, then the program will dynamically modify the loop instruction in the memory playlist structure. This is how the device playing the playlist knows how many times to play the chime. The variable *usHour* is assigned the valid hour in the UpdateTheClock procedure.

```
      /*
       * Set the playlist to loop for the number of hours to
       * be played.  The hour value will be placed into operand one
       * of the loop instruction playlist structure.*/

           apltPlayList[ HOUR_PLAYLIST ][ LOOP_LINE ].ulOperandOne =
               ((ULONG) (usHour));
```

Manipulations that entail atomic, or uninterruptible, operations are acceptable techniques for achieving special effects with playlists. For example, the modification of the 32-bit operation code of an instruction is considered to be an atomic operation. Changing BRANCH to NOP, NOP to BRANCH, or changing the loop count value of a LOOP instruction are examples of atomic operations that produce special effects.

Because the playlist processor is asynchronously interpreting instructions, care must be taken to avoid a situation where unexpected results can occur. For example, suppose a situation exists that enables the loop count field to be modified by the application at the same time that the LOOP instruction is being executed by the playlist processor. The playlist processor can overwrite the application modification with a loop count value based on the execution of the LOOP instruction.

More extensive modifications (particularly those to pointer and length parameters of DATA instructions) should also be performed with caution. You do not want the playlist processor to gain control of a partially modified instruction.

As a rule, it is not a good idea to modify a DATA instruction unless you are sure the playlist processor cannot interpret the instruction during the modification process; that is, you know the instruction is currently unreachable by the processor. One way to determine the location of the processor is with the MESSAGE instruction. For example, suppose you precede a LOOP instruction with a MESSAGE instruction. When the message is returned to your application, you know the processor is busy with the loop and you can safely modify any DATA instructions outside the loop.

After DATA instructions have been modified, BRANCH or NOP instructions can be changed to direct playlist interpretation to the modified section of the playlist.

Using a Playlist for Recording

Memory playlists can be used in recording scenarios. For example, MESSAGE instructions are useful for notifying the application of the progress of a recording operation. A message can be sent each time the filling of a buffer is completed.

Encasing several DATA statements inside a loop using either a LOOP or BRANCH instruction allows the playlist to function as a simple circular buffering scheme. The following is an example of circular buffering:

```
0:    NOP
1:    DATA...
2:    MESSAGE...
3:    DATA...
4:    MESSAGE...
5:    DATA...
6:    MESSAGE...
7:    BRANCH 0
```

If the playlist processor executes an EXIT statement while recording, this means the playlist is full. This situation is similar to a disk becoming full during a recording operation. An ERROR_END_OF_PLAYLIST error is generated by the playlist processor in the streaming subsystem. As the recording operation continues, additional data is lost. The media driver being used returns the MCIERR_TARGET_DEVICE_FULL to the application.

LOOP, CALL, and RETURN instructions are used for iterative playing operations. These instructions are not appropriate for a recording scenario because recorded data residing in the buffer would be overwritten by each iteration.

Editing Operations

Applications can perform editing operations using both the PM clipboard and user-defined buffers. This enables applications to share data with other applications by way of the clipboard or to quickly retrieve data into user-defined buffers. Messages used for editing include MCI_COPY, MCI_CUT, MCI_DELETE, MCI_PASTE, MCI_REDO, and MCI_UNDO.

Clipboard and Resource Formats

Applications can imbed multimedia data into program resources and interchange that data through the clipboard.

The following clipboard and resource format types are defined in the OS2MEDEF.H file:

- CF_RMID and RT_RMID represent RIFF data that has a RMID chunk or regular MIDI with an "MT" header. This is a special case.
- CF_RIFF and RT_RIFF represent RIFF data including all of the headers.
- CF_WAVE and RT_WAVE represent RIFF data that has a WAVE chunk. This is a specific form of RIFF.
- CF_AVI and RT_AVI represent RIFF Audio/Video Interleaved (AVI) data. This is a specific form of RIFF. An entire video file is

placed in the clipboard.

The following code fragment is an example of retrieving digital audio information from the clipboard. Source code for ADMCEDIT is located in the \TOOLKIT\SAMPLES\MM\ADMCT subdirectory.

```
{
HAB            habClipboard;
HMQ            hmq;

HMMIO          hmmioMem;           /* handle to memory file   */

MMIOINFO       mmioinfo;           /* info struct for memory file */

ULONG          ulFormatInfo = 0;
ULONG          rc;
PULONG         pulDataSize;

LONG           lBytesRead;
LONG           lReturnCode;

MMAUDIOHEADER  mmaudioheader;
MMAUDIOHEADER  mmaudioheaderTemp;

PSZ            pTempBuffer;
PVOID          pNewBuffer;

  habClipboard = WinInitialize( 0 );
  if ( !habClipboard )
     {
     return ( MCIERR_CLIPBOARD_ERROR );
     }

  hmq = WinCreateMsgQueue( habClipboard, 0 );

  if ( !hmq )
     {
     fCreatedMQ = FALSE;
     }

/*  habClipboard = WinQueryAnchorBlock( HWND_DESKTOP ); */

  /*****************************************************************
  * Check to see if there is a wave (CF_WAVE is the defined type) in
  * the clipboard.
  ******************************************************************/

  rc = WinQueryClipbrdFmtInfo( habClipboard,
                          CF_WAVE,
                          &ulFormatInfo );


  if ( !WinOpenClipbrd( habClipboard ) )
     {
     WinCloseClipbrd( habClipboard );
     return ( MCIERR_CLIPBOARD_ERROR );
     }

  pTempBuffer = ( PSZ ) WinQueryClipbrdData( habClipboard, CF_WAVE );

  if ( !pTempBuffer )
     {
     WinCloseClipbrd( habClipboard );
     return ( MCIERR_CLIPBOARD_ERROR );
     }

  /*****************************************************************
  * We need to find out how much data is in the file.  Retrieve
  * the length of the RIFF chunk.
  ******************************************************************/
  pulDataSize = ( PULONG ) pTempBuffer + 1;


  memset( &mmioinfo, '\0', sizeof( MMIOINFO ) );

  /*****************************************************************
  * Prepare to open a memory file--the buffer * in the clipboard
  * contains the actual RIFF file which the WAVE IOProc already knows
  * how to parse--use  it to retrieve the information and keep the MCD
  * from file-format dependence.
  ******************************************************************/

  mmioinfo.fccIOProc = mmioFOURCC( 'W', 'A', 'V', 'E' ) ;
  mmioinfo.fccChildIOProc = FOURCC_MEM;

  rc = CheckMem ( (PVOID) pulDataSize,
                    sizeof ( ULONG ),
                    PAG_READ | PAG_WRITE );

  if (rc != MCIERR_SUCCESS)
     {
     WinCloseClipbrd( habClipboard );
     return (MCIERR_OUT_OF_MEMORY );
     }


  mmioinfo.cchBuffer = ( *pulDataSize) + 8;
  mmioinfo.pchBuffer = pTempBuffer;


  hmmioMem = mmioOpen( NULL,
                       &mmioinfo,
                       MMIO_READ );

  if ( !hmmioMem )
     {
     WinCloseClipbrd( habClipboard );
     return ( mmioinfo.ulErrorRet );
     }
```

Audio Media Driver Clipboard Commands

The data that the buffer or clipboard contains will be defined differently for each media driver.

The following formula is necessary to allocate memory for digital audio clipboard operations:

(BitsPerSample/8) x (SamplesPerSecond) x (Channels) x (Seconds)

Therefore, if an application is to copy 20 seconds of a 16-bit, 44 kHz, mono file into the clipboard using the caller's buffer; 1,764,000 bytes ((16/8) x (44100) x (1) x (20)) must be allocated and placed in the *pBuff* field of the MCI_EDIT_PARMS structure. For MCI_CUT, MCI_COPY, and MCI_PASTE, if MCI_TO_BUFFER or MCI_FROM_BUFFER is passed in, then the *pBuff* field should contain a valid pointer.

MCI_STATUS_CLIPBOARD returns MCI_TRUE if digital audio is in the clipboard; otherwise it returns MCI_FALSE. MCI_CUT removes the specified range and places the data in the buffer or clipboard. The position of the media will either be the from position if MCI_FROM is specified or the previous position if MCI_FROM is not specified. If the buffer is not large enough for the data an MCIERR_INVALID_BUFFER is returned. The units of MCI_FROM and MCI_TO must be supplied in the currently selected time format. If neither MCI_FROM or MCI_TO are specified, the operation will start from the current file position and continue to the end of the file. If audio data is already in the clipboard, it will be overwritten.

Note: The clipboard contents are emptied before the cut occurs.

MCI_COPY copies the specified range and places the data in the buffer or clipboard. The position of the media remains the same as it was before the copy operation.

MCI_PASTE deletes the selected range if the differences between the from and to position are greater than zero, then inserts the data provided in the buffer or clipboard. The media position will be at the end of what was pasted into the file. If neither MCI_FROM or MCI_TO are specified, MCI_PASTE inserts the clipboard contents at the current position. MCI_CONVERT_FORMAT converts the data that was in the clipboard to the destination file format. The following data format conversions can be performed:

- 16-bit to 8-bit resolution/8-bit to 16-bit resolution
- 11.025 kHz, 22.05 kHz, and 44.1 kHz to any of the following sampling rates: 11.025 kHz, 22.05 kHz, or 44.1 kHz

- mono to stereo/stereo to mono

Note: The MCI_CONVERT_FORMAT flag supports only the Pulse Code Modulation (PCM) format. The data format conversion can take a while to complete. If the notify flag is specified, the application is notified when the conversion is completed.

The following code fragment shows an example of the use of MCI_COPY and MCI_PASTE.

```
ULONG weMciCopy( HWND hwnd, ULONG ulMarkedStartBytes,
                 ULONG ulMarkedEndBytes,  USHORT usDeviceID )
   {
   ULONG          ulFlags;
   MCI_EDIT_PARMS mcieditstr;
   ULONG          ulResult;

   ulResult = 0L;

   /*
    * First, set all fields of the MCI_EDIT_PARMS structure to 0.
    */
   memset( &mcieditstr, '\0', sizeof(MCI_EDIT_PARMS) );

   /*
    * The flags are NOTIFY, FROM, and TO.
    */
   ulFlags = 0L;
   ulFlags |= MCI_NOTIFY |
              MCI_FROM |
              MCI_TO;

   mcieditstr.ulCallback = (ULONG)hwnd;

   /*
    * Set the from and to items to the beginning and end
    * of the selected area.
    */
   mcieditstr.ulFrom = ulMarkedStartBytes;

   mcieditstr.ulTo = ulMarkedEndBytes;

   ulResult = mciSendCommand( usDeviceID,
                              MCI_COPY,
                              ulFlags,
                              (ULONG)&mcieditstr,
                              0 );

   return( ulResult );
   {

ULONG weMciPaste( HWND hwnd, ULONG ulMarkedStartBytes,
                  ULONG ulMarkedEndBytes,  USHORT usDeviceID )
   {
   ULONG          ulFlags;
   MCI_EDIT_PARMS mcieditstr;
   ULONG          ulResult;

   ulResult = 0L;

   /*
    * First, set all fields of the MCI_EDIT_PARMS structure to 0.
    */
   memset( &mcieditstr, '\0', sizeof(MCI_EDIT_PARMS) );

   mcieditstr.ulCallback = (ULONG)hwnd;

   ulFlags = 0L;

   /*
    * If there is an area of wave selected, then the flags are NOTIFY,
    * FROM, TO, and CONVERT_FORMAT.
    */
   if( ulMarkedEndBytes > ulMarkedStartBytes )
      {
      ulFlags |= MCI_NOTIFY |
                 MCI_FROM |
                 MCI_TO |
                  MCI_CONVERT_FORMAT;

      /*
       * Set the from and to items to the beginning and end
       * of the selected area.
       */
      mcieditstr.ulFrom = ulMarkedStartBytes;

      mcieditstr.ulTo = ulMarkedEndBytes;
      }
   else
      {
      /*
       * Otherwise, nothing in the wave is selected so the flags are
       * only NOTIFY and CONVERT_FORMAT.
       */
      ulFlags |= MCI_NOTIFY |
                 MCI_CONVERT_FORMAT;

      /*
       * Because this is a paste operation without FROM/TO,
       * we have to SEEK so that the media position is set
       * to the place that we want to paste.
       */
      if( ulResult = weMciCall( hwnd,
                                MCI_SEEK ) )
         return( ulResult );
      }
   ulResult = mciSendCommand( usDeviceID,
                              MCI_PASTE,
                              ulFlags,
                              (ULONG)&mcieditstr,
                              0 );

   return( ulResult );
   {
```

The following is an example of using the command string interface with editing commands to create a repeating sound.

```
open test.wav alias a wait
copy a from 0 to 3000 wait
seek a to end
paste a wait
paste a wait
paste a wait
```

Device Sharing By Applications

The multimedia system supports sharing of physical devices among multiple applications. If a device is capable of being shared; that is, if it can maintain state information, the system can establish a unique device state, much like a Presentation Manager device context, for each application that uses the device.

The scope of a device state is defined by each device. The state of a simple device like the digital video player contains information about the current frame position, whether the device is playing or stopped, what its current playback speed is set to, and so on. The state of a compound device can include the name of the currently selected file, RIFF object, and playback position.

Media devices vary in their ability to support multiple device contexts concurrently. The different types of device use that are supported by media devices are:

- Fixed single-context
- Dynamic single-context
- Limited multiple-context
- Unlimited context

The following table contains descriptions and examples of these device use types.

Context Use Type	Description
Fixed single-context	A fixed single-context device can establish only one device context. The state of a fixed single-context device cannot be queried or set by software. An example of a fixed single-context device is a video cassette recorder that does not report the tape position to the driver.
Dynamic single-context	A dynamic single-context device is serially shareable. That is, the device can be used by only one application at a time but can be passed from one application to another. A device state for each application is saved and restored appropriately. This is the most common concurrent use type for a media device. An example of a dynamic single-context device is a CD-ROM player.
Limited multiple-context	A limited multiple-context device can establish multiple device contexts, but the number of device contexts is limited by the physical device. An example of a limited multiple-context device is a 4-channel amp-mixer audio device, which can concurrently support any of the following multiple-contexts: Four monaural contexts, two stereo contexts, and one stereo and two monaural contexts.
Unlimited context	An unlimited context device can support an arbitrary number of concurrent device contexts. The number of concurrent contexts is limited only by the resource limits of the system.

Getting Control of a Shared Device

The MM_MCIPASSDEVICE message sent with WinPostMsg by the multimedia system to applications and the MCI_ACQUIREDEVICE message sent by applications with mciSendCommand to the multimedia system provide a device-sharing scheme for the OS/2 multimedia environment.

To participate in device sharing, an application issues MCI_OPEN with the MCI_OPEN_SHAREABLE flag set. The system then attempts to acquire the device for the application. The application must wait until it receives the asynchronous MM_MCIPASSDEVICE message to gain control of the device. The multimedia system sends the MM_MCIPASSDEVICE message to inform the application that the device context is becoming active (MCI_GAINING_USE).

Before an application receives an MM_MCIPASSDEVICE message with an event of MCI_GAINING_USE, it can make inquiries about the device and the media. MCI_STATUS, MCI_GETDEVCAPS, MCI_INFO, and MCI_CLOSE commands can be sent to an inactive device context.

Note: If your application has set an MCI_NOTIFY flag on the open request, notification will be posted to the application before the MM_MCIPASSDEVICE message is sent. However, if the application message queue has other messages already queued, it is possible that the application may receive the MM_MCIPASSDEVICE message before it receives the notification message.

The active instance of the application remains active until the application returns from the WinPostMsg (MCI_LOSING_USE). This guarantees that the application has an active device context until it returns from WinPostMsg. If the application receives an MM_MCIPASSDEVICE message with an event of MCI_GAINING_USE, it should return immediately. The following code fragment illustrates the device sharing architecture from the Clock Sample program.

```
  /*
   * The next two messages are handled so that the Clock application
   * can participate in device sharing.  Because it opens the devices
   * as shareable devices, other applications can gain control of the
   * devices.  When this happens, we will receive a pass device
   * message.  We keep track of this device passing in the fPassed
   * boolean variable.
   * If we do not have access to the device when we receive an
   * activate message, then we will issue an acquire device command
   * to gain access to the device.
   */

case MM_MCIPASSDEVICE:
   if (SHORT1FROMMP(mp2) == MCI_GAINING_USE)
   {
      fPassed = FALSE;               /* Gaining control of device   */
   } else
   {
      fPassed = TRUE;                /* Losing control of device    */
   }
   return( WinDefSecondaryWindowProc( hwnd, msg, mp1, mp2 ) );

case WM_ACTIVATE:

   /* We use the WM_ACTIVATE message to participate in device sharing.
    * We first check to see if this is an activate or a deactivate
    * message (indicated by mp1). Then, we check to see if we've
    * passed control of the device that we use.  If these conditions
    * are true, we issue an acquire device command to regain
    * control of the device, because we're now the active window on
    * the screen.
    *
    * This is one possible method that can be used to implement
    * device sharing. For applications that are more complex
    * than this sample program, developers may wish to take
    * advantage of a more robust method of device sharing.
    * This can be done by using the MCI_ACQUIRE_QUEUE flag on
    * the MCI_ACQUIREDEVICE command.
    */

   /*
    * First we check to see if we've passed control of the device
    */

if ((BOOL)mp1 && fPassed == TRUE) {

   mciGenericParms.hwndCallback =  hwnd;

   ulError = mciSendCommand( mciOpenParameters.usDeviceID,
                             MCI_ACQUIREDEVICE,
                             (ULONG)MCI_NOTIFY,
                             (PVOID) &mciGenericParms,
                             (USHORT)NULL);
   if (ulError)
   {
     ShowAMessage(acStringBuffer[IDS_NORMAL_ERROR_MESSAGE_BOX_TEXT-1];
                  IDS_CHIME_FILE_ERROR,  /* ID of message      */
                  MB_OK | MB_INFORMATION |
                  MB_HELP | MB_APPLMODAL |
                  MB_MOVEABLE );         /* Style of msg box.  */
   }
 }
 return( WinDefSecondaryWindowProc( hwnd, msg, mp1, mp2 ) );
```

Regaining Control of a Shared Device

An application regains control of a shared device by issuing the MCI_ACQUIREDEVICE message with mciSendCommand after it has received a WM_ACTIVATE message. The application receives a WM_ACTIVATE message whenever its frame window is activated or deactivated by user selection. The time for the application to regain control of a shared device is during the period its window is activated. A "greedy" application that grabs back a device as soon as it loses it defeats the purpose of the WM_ACTIVATE message processing scheme, which is to give control of a shared device to the application with which the user is interacting.

Only dynamic single-context and limited multiple-context devices are acquired by applications. The MCI_ACQUIREDEVICE function does not perform any function for fixed single-context and unlimited-context devices, because device contexts are not saved or restored for these classes of devices.

To better understand the allocation of resources to multiple device contexts, imagine a stack of device contexts. The physical device is associated with the topmost device context on the stack. Whenever a device context is opened, it is placed on top of the stack, and the physical device is associated with the new device context. When MCI_ACQUIREDEVICE is issued for a particular device context, that device context moves to the top of the stack, and the physical device is associated with the existing device context. Closing a device context removes it from the stack.

Queued Acquire Command

Setting the MCI_ACQUIRE_QUEUE flag of the MCI_ACQUIREDEVICE message enables the message to be queued and executed as soon as device resources become available. An application can issue an MCI_ACQUIREDEVICE message and, at a later point, the device context becomes active. This is true if either the MCI_NOTIFY or MCI_WAIT flag is specified. If the MCI_WAIT flag is specified, the calling thread is blocked until the device context becomes active. If the MCI_ACQUIREDEVICE request can be satisfied immediately, the command is not queued.

The **acquire** command can be used to acquire a device instance when the resource becomes available:

```
open music1.wave alias wave1 shareable wait
play wave1 notify
.
.
.
** During this time a losing use message is received **
** and this instance becomes inactive.               **
.
.
.
acquire wave1 queue notify
```

If an MCI_ACQUIREDEVICE is queued and an application issues MCI_RELEASEDEVICE or MCI_CLOSE for that instance, the queued MCI_ACQUIREDEVICE message is canceled.

Releasing the Resource

The release resource command is used in conjunction with the queued acquire command. An application can release a device instance from the active state and make the next available inactive device instance active by setting the MCI_RETURN_RESOURCE flag of the MCI_RELEASEDEVICE message. When a device instance no longer needs its resources, the device instance can give up the resource to another device requesting the resources (with MCI_ACQUIRE_QUEUE).

The **release** command as shown in the following example can be used to release exclusive hold on a device.

```
open waveaudio alias wave2 shareable wait
acquire wave2 exclusive wait
record wave2 notify
.
.
.
** Open the device exclusively to avoid interruptions **
** during recording.                                   **
.
.
.
stop wave2 wait
release wave2 return resource wait
```

The device instance will not be made active again unless an application issues an MCI_ACQUIREDEVICE message for this device context. This function is ignored if the instance is already in an inactive state. The instance remains active if the resource used by this instance is not required by any other instance.

Using a Device Exclusively

There are times when an application must retain control of the physical resource, such as the during the duration of a recording operation or when the application needs to establish specific settings for the device context. The application can retain control by *not* specifying the shareable flag with the open request or by setting the MCI_EXCLUSIVE flag of the MCI_ACQUIREDEVICE message. When a device has been acquired for exclusive use, other applications cannot acquire the device until the application using the device closes it or releases it from exclusive use with the MCI_RELEASEDEVICE message. When an application releases a device from exclusive use, it does not lose use of the device until another application acquires it.

When an application needs to acquire a device context for exclusive use without acquiring the entire device resource, the application can set the MCI_EXCLUSIVE_INSTANCE flag of the MCI_ACQUIREDEVICE message. This flag prevents the device context from being made inactive unless the application using the device issues the MCI_CLOSE or MCI_RELEASEDEVICE message.

The MCI_EXCLUSIVE_INSTANCE and MCI_EXCLUSIVE flags are mutually exclusive. An instance can be in one of three sharing states:

- Instance exclusive
- Device exclusive
- Fully shareable

Using the MCI_EXCLUSIVE_INSTANCE flag places an instance in an instance-exclusive sharing state. Using the MCI_EXCLUSIVE flag places an instance in a device-exclusive sharing state. The MCI_RELEASEDEVICE message places an instance in a fully shareable state.

Device Groups

When an OS/2 multimedia application needs to control more than one device at a time, it uses the MCI_GROUP message to group device contexts. The MCI_GROUP_MAKE and MCI_GROUP_DELETE flags are used to make and delete groups. MCI_GROUP_MAKE ties several device instances together so that a single command sent to the group by an application is actually sent to each device instance in the group by the multimedia system. This flag can be combined with any of the other MCI_GROUP flags except MCI_GROUP_DELETE in which case an MCIERR_FLAGS_NOT_COMPATIBLE error code is returned. Device instances must have been previously opened but can be in any mode (such as, playing, stopped, or paused) for this message to be successful. If one or more device IDs are invalid, the MCIERR_INVALID_DEVICE_ID error code is returned. If a device ID or alias refers to a device in another group, the MCIERR_ID_ALREADY_IN_GROUP error code is returned.

Once a group has been made, certain command messages sent to the group ID (or alias name) are sent to each device making up that group. Command messages that support groups are:

MCI_ACQUIREDEVICE
MCI_CLOSE
MCI_CUE
MCI_PAUSE
MCI_PLAY
MCI_RECORD
MCI_RELEASEDEVICE
MCI_RESUME
MCI_SEEK
MCI_SET
MCI_STOP

Note: Commands sent to a group must use the MCI_NOTIFY flag.

To end a group association, an application uses the MCI_GROUP_DELETE flag of the MCI_GROUP message. None of the other flags of the MCI_GROUP message can be combined with MCI_GROUP_DELETE because the only information required by this flag is a group ID. If any other flags are supplied an MCIERR_FLAGS_NOT_COMPATIBLE error code is returned. The MCIERR_INVALID_GROUP_ID error code is returned if an application passes an invalid ID. When a device in a group is closed, it is removed from the group. When the last device in a group is closed, the group is automatically deleted.

Applications can use the MCI_GROUP_ALIAS flag to refer to a group by a name rather than a group ID for use with the mciSendString interface. This flag can only be used with an MCI_GROUP_MAKE flag; the given alias is used to refer to the new group. If the alias is already in use, an MCIERR_DUPLICATE_ALIAS error code is returned. Each string group "make" command must include an alias so the group can later be referred to. The alias follows the **group** command as shown by the following syntax:

```
group grp1 make (wave1 cd1) wait
```

This defines the alias to be "grp1". The list of device names (members of the group) is delimited by parenthesis and separated by spaces and optional quotation marks. The following syntax is used to delete this group:

```
group grp1 delete wait
```

Duet Player Sample Program Example

The following code fragment illustrates the creation of a device group in the Duet Player I sample program. An array is filled with the IDs of opened devices to be associated in the group. The application then calls MCI_GROUP to create the group and return a handle.

```
  /* If this is the first time through this routine, then we need to
   * open the devices and make the group.
   *
   * On subsequent calls to this routine, the devices are already open
   * and the group is already made, so we only need to load the
   * appropriate files onto the devices.
   */
  {
  /*
   * Open one part of the duet. The first step is to initialize an
   * MCI_OPEN_PARMS data structure with the appropriate information,
   * then issue the MCI_OPEN command with the mciSendCommand function.
   * We will be using an open with only the element name specified.
   * This will cause the default connection, as specified in the
   * MMPM.INI file, for the data type.
   */
  mopDuetPart.hwndCallback   =  hwnd;     /* For MM_MCIPASSDEVICE   */
  mopDuetPart.usDeviceID     = (USHORT) NULL; /* this is returned   */
  mopDuetPart.pszDeviceType  = (PSZ) NULL;    /* using default conn.*/
  mopDuetPart.pszElementName = (PSZ) aDuet[sDuet].achPart1;

  ulError = mciSendCommand( (USHORT) 0,
                            MCI_OPEN,
                            MCI_WAIT | MCI_OPEN_ELEMENT |
                            MCI_OPEN_SHAREABLE | MCI_READONLY,
                            (PVOID) &mopDuetPart,
                            UP_OPEN);

  if (!ulError)  /* if we opened part 1 */
  {
     usDuetPart1ID = mopDuetPart.usDeviceID;

     /*
      * Now, open the other part.
      */
     mopDuetPart.pszElementName    = (PSZ)   aDuet[sDuet].achPart2;

     ulError = mciSendCommand( (USHORT) 0,
                               MCI_OPEN,
                               MCI_WAIT | MCI_OPEN_ELEMENT |
                               MCI_OPEN_SHAREABLE | MCI_READONLY,
                               (PVOID) &mopDuetPart,
                               UP_OPEN);

  if (!ulError)  /* if we opened part 2 */
  {
     usDuetPart2ID = mopDuetPart.usDeviceID;

 /*
  * Now we need to create a group.  To do this,
  * we need to fill an array with the IDs of the already open
  * devices that we want to group.  Then we call MCI_GROUP to
  * create the group and return a handle to it.
  */

 ulDeviceList[0] = (ULONG)usDuetPart1ID;
 ulDeviceList[1] = (ULONG)usDuetPart2ID;

 mgpGroupParms.hwndC  lback = (HWND) NULL;     /* Not needed -
                                                  we're waiting     */
 mgpGroupParms.ulNumDevices = NUM_PARTS;       /* Count of devices  */
 mgpGroupParms.paulDeviceID = (PULONG)&ulDeviceList; /* Array of
                                                        devices     */
 mgpGroupParms.ulStructLength = sizeof (mgpGroupParms);

ulError = mciSendCommand( (USHORT) 0,
                           MCI_GROUP,
                           MCI_WAIT | MCI_GROUP_MAKE|
                           MCI_NOPIECEMEAL,
                           (PVOID) &mgpGroupParms,
                           UP_GROUP);

 fFirstPlay = FALSE;
```

Resource Allocation

An application avoids piecemeal resource allocation problems by setting the MCI_NOPIECEMEAL flag of the MCI_GROUP message. This flag specifies that the associated group is treated as a whole rather than several separate instances. If one of the device instances becomes inactive then all the instances in the group will go inactive. This flag can only be combined with the MCI_GROUP_MAKE flag as it specifies the nature of the group to be created. If the MCI_NOPIECEMEAL flag is set during creation and one or more of the instances in the list of IDs or aliases is already inactive, the entire group will go inactive and each device in the group saves its state. The device contexts in the group can subsequently be restored by passing the group device context ID with the MCI_ACQUIREDEVICE message, using mciSendCommand.

If the MCI_NOPIECEMEAL flag is not specified and devices are lost, the application retains control over the remaining devices in the group, unless one of the lost devices is the master of the group. When the master of a group of devices is lost, the group is lost.

Event Synchronization

Applications can perform event synchronization by taking an action at a specified point during the playback of a data object. There are two ways an application can do this:

- The application can request to be notified when a specified point in playback is encountered by sending the MCI_SET_CUEPOINT message to the multimedia system. When this cue point is encountered, the multimedia system sends an MM_MCICUEPOINT message to the application.
- The application can request notification on a periodic basis, based on time or position, by sending the MCI_SET_POSITION_ADVISE message to the multimedia system. As each time period (or position) specified passes, the multimedia system sends an MM_MCIPOSITIONCHANGE message to the application.

Cue Points

Cue points are discrete locations or time positions in a media device. When a device encounters a time position associated with a cue point, a message is returned to the application window handle that is specified to receive the cue point messages. Cue points are maintained as part of a device context, so setting a cue point in one device context will not cause cue point messages to be generated for other device contexts.

Applications specify cue points for a device with the MCI_SET_CUEPOINT message. A cue point is identified by its location; setting a cue point "on" sets a cue point at the specified location, and setting a cue point "off" removes the cue point. Because cue points are identified by location, only one cue point can be set at a specified location in the media. Therefore, setting a cue point at a location where a cue point is already set causes the second MCI_SET_CUEPOINT to fail and to return the error MCIERR_DUPLICATE_CUEPOINT. Cue points can be set at any valid location in the media, regardless of current media position. If a device is currently playing at 2:00 (two minutes), and a cue point is set at 1:00 (one minute) in the media, and the device is subsequently seeked and played from the beginning, the cue point message will be generated when the device passes the 1:00 point in the media.

Cue points are persistent. That is, they remain set after they are encountered. The device will generate cue point messages whenever the cue point location is encountered, which may be many times if the device is seeked or played repeatedly. Cue points are encountered only when a device is playing or recording. If a device is seeked from its current position to some new position, cue points set at locations between the old and new position are not encountered during the seek operation, and no cue point messages are generated.

Because cue points can be set only within the valid range of a media element, cue points cannot be set when a file is not loaded. All cue points for a device context are removed when a new file element is loaded.

Cue points also can be created as part of a media element. In the case of cue points imbedded directly in a media element, the MCI_SET_CUEPOINT message performs no function. Imbedded cue points always result in cue point messages being returned when they are encountered. The user parameter value returned on the cue point message varies from one media data type to another and should be set to a meaning that is significant to the application.

When a cue point is encountered, an MM_MCICUEPOINT message is sent to the window specified by the *hwndCallback* field of the MCI_CUEPOINT_PARMS data structure passed with the MCI_SET_CUEPOINT message. The MM_MCICUEPOINT message parameters contain the device ID of the device context that generated the cue point message, as well as the media position and an additional application-defined parameter that can be specified when the cue point is set. Although the media position specified by the application on the MCI_SET_CUEPOINT message is in the currently set device units, the media position returned on the MM_MCICUEPOINT message is always in MMTIME units. MMTIME units are used because the time format set when the cue point is set might not be the same time format set when the cue point is encountered.

The maximum number of cue points that can be set in a device context is defined by the implementation of the logical device. Devices generally support up to 20 cue points per device context.

Position Advises

In addition to notification messages at discrete locations in the media, periodic notification of elapsed media time can also be requested. These periodic messages, referred to as "position advise" messages, are requested for a device context based on a specified time interval. Position advise messages are requested by issuing the MCI_SET_POSITION_ADVISE message to a device context as shown in the following code fragment.

```
        MCI_OPEN_PARMS   mop;
static  MCI_PLAY_PARMS   mpp;      /* parms for MCI_PLAY         */
static  MCI_POSITION_PARMS mppPos; /* parms for
                                      MCI_SET_POSITION_ADVISE */

iState = ST_PLAYING;           /* Set state to reflect play mode   */

mppPos.hwndCallback = hwndMainDlg;
mppPos.ulUnits    = 1500;      /* Request position advise messages */
mppPos.usUserParm = usPositionUP;
mppPos.Reserved0  = 0;
mciSendCommand    ( mop.usDeviceID,
                    MCI_SET_POSITION_ADVISE,
                    MCI_NOTIFY | MCI_SET_POSITION_ADVISE_ON,
                    (PVOID) &mppPos,
                    UP_POSITION );
```

This causes MM_MCIPOSITIONCHANGE messages to be returned to the application window specified in the MCI_POSITION_PARMS structure at the requested frequency as media time passes in the device context. Only one position advise frequency may be active for a device context, and having position advise notification active in one device context does not cause messages to be generated in other device contexts. Position advise messages can be set only when a device element is loaded in the device context, and are reset when a new device element is loaded.

Like MM_MCICUEPOINT messages, MM_MCIPOSITIONCHANGE message parameters contain the device ID of the device context that generated the position advise message, as well as the media position and an additional application-defined parameter that can be specified when the position advise notification is requested. Although the media position interval (frequency) specified by the application on the MCI_SET_POSITION_ADVISE message is in the currently set device units, the media position returned on the MM_MCIPOSITIONCHANGE message is always in MMTIME units. MMTIME units are used because the time format set when the position advise notification is set might not be the same time format set when the position advise notification messages are returned.

Position advise notifications are generated only during playback or recording. MM_MCIPOSITIONCHANGE messages are usually not generated during seek operations initiated by the application. The exception is a device, such as a tape recorder, that has a discernible position during the seek operation. A device like this can generate position advise messages as the media is traversed, to indicate the progress of the seek operation.

The following code fragment shows how the Caption Sample application handles the MM_MCIPOSITIONCHANGE message. When the Caption Sample application receives a position change message, it updates its media position slider arm allowing the application to advance the media position slider smoothly as the audio plays.

```
case MM_MCIPOSITIONCHANGE:
 /*
  * This message will be returned (in MMTIME) to the application
  * whenever the audio position changes. This time will be used to
  * increment the audio position slider. This message is only
  * generated during playback.
  */
 if ( eState == ST_PLAYING )
```

```
{
   ulTime = (ULONG) LONGFROMMP(mp2);

   /*
    * Get the new slider arm position and set it.
    */
   sArmPosition =
      (SHORT) ( ( ulTime * ( sAudioArmRange - 1) ) / ulAudioLength );
   WinSendMsg(
      hwndAudioSlider,
      SLM_SETSLIDERINFO,
      MPFROM2SHORT( SMA_SLIDERARMPOSITION, SMA_RANGEVALUE ),
      MPFROMSHORT( sArmPosition ));
}
return 0;
```

System Values

The OS/2 multimedia system provides a number of system-wide values that can be queried and set by applications. Because OS/2 multimedia applications such as Volume Control and Multimedia Setup allow users to set system values, it is recommended that applications only query the settings users have selected. The following table describes the system values that can be queried and set using mciQuerySysValue and mciSetSysValue.

System Value	Description
MSV_CLOSEDCAPTION	Query or set the current state of a captioning flag. By querying the setting of this flag, an application can determine whether to display text along with audio, for example, for a hearing-impaired user.
MSV_MASTERVOLUME	Query or set the current master audio level. This value acts as a "multiplier" of the individual volume levels of each device context, allowing one application to control the volume for a number of open devices or elements.
MSV_HEADPHONES	Reserved for future use.
MSV_SPEAKERS	Reserved for future use.
MSV_WORKPATH	Query or set the directory for storing of temporary files by the media driver. This value can be used to point to, for example, a directory on the hard disk that holds waveform data from a recording operation.
MSV_SYSQOSERRORFLAG	Query the Quality of Service (QOS) error flag. By querying this flag, an application can determine an error occuring during band-width reservation.
MSV_SYSQOSVALUE	Query or set the QOS specification value. This system-wide Quality of Service (QOS) specification value is used for band-width reservation (for example, bytes per second) over the network.

The following code fragment demonstrates how to obtain the multimedia work path.

```
CHAR szWorkpath[CCHMAXPATH] ;   /* Work path for temporary files */
```

```
if ( mciQuerySysValue( MSV_WORKPATH, szWorkPath ) )
  {
  /* mciQuerySysValue was successful, szWorkPath now */
  /* contains the multimedia workpath                */
  }
```

Clock Sample Program Caption Query

When it is time to chime the clock, the Clock Sample program checks the system captioning flag to determine whether or not it should display a visual chime while the audio chime is playing. The Clock program sets the global variable *fClosedCaptionIsSet* to store the value of the system captioning flag.

```
 /*
  * If the Captioning Flag indicates that the bell should be
  * animated (swung), the region of the presentation space
  * that contains the bell bitmap is to be invalidated so that a
  * WM_PAINT will be sent to draw the bells.
  */
 mciQuerySysValue( MSV_CAPTION, (PVOID)&fCaptionIsSet );
```

Multimedia Logical Devices

OS/2 multimedia represents audio adapters, CD-ROM drives, videodiscs and other real hardware devices as logical media devices that are managed by the Media Device Manager (MDM). Media devices are a logical representation of the function available from either a real hardware device, software emulation in combination with real hardware, or pure software emulation. The actual implementation is not relevant to an application, because the multimedia system provides device independence with the mciSendCommand and mciSendString interfaces.

The following logical devices are currently supported in this release of OS/2. Additional media devices may be available from IBM or from other companies as OS/2 multimedia is completely extensible at all levels.

- Amplifier mixer
- Waveform audio
- MIDI sequencer
- CD audio
- CD-XA
- Videodisc
- Digital video

Frequently there is a one-to-one correspondence between a real hardware device, such as a CD-ROM drive and its associated media device. Other hardware may be represented as multiple logical devices. For example, a multi-function audio adapter can be represented as waveform audio, MIDI sequencer, and amplifier-mixer media devices.

The following sections describe the function and typical use of each media device, plus the software model presented to the application developer.

Multimedia Information and OS/2 Multimedia Connectors

A *connector* is a software representation of the physical way in which multimedia data moves from one device to another. Simple examples are the headphone jack on a CD-ROM player, or the speakers jack on an audio adapter. If an audio card has both a speaker and a line OUT jack, it is desirable to let an application choose the destination of the audio, while remaining independent from the actual hardware implementation.

OS/2 multimedia connectors provide this function by allowing an application to query which connectors are supported by a logical device, and manipulate whether or not information is flowing through the connector. The connectors for a logical device can be accessed either by number or by a symbolic connector type. When specifying a symbolic type such as *microphone* or *line IN*, a number can also be specified to select the first connector, second connector, and so on, of that specific connector type. The MCI_CONNECTORINFO message can be used to determine which connectors are supported by a device, whereas the MCI_CONNECTOR message can be used to enable, disable, or query the state of a particular connector.

Although connectors are typically associated with the representation of externally visible audio and video jacks on multimedia equipment, another category of connectors can represent the flow of information within a computer. For example, a connector on an audio adapter can be attached to the internal PC speaker. A more subtle example is the flow of digital audio information into an audio adapter. This information could come from a file, system memory, or another device. Connectors of this category are referred to as *stream* connectors to convey the idea of a logical stream of information flowing from one device to another.

Connector Types

Each connector is defined by specific type or name, so that applications can make requests symbolically instead of using an absolute connector number that is device dependent. If an application specifies only a connector type, then the default connector of that type is selected on the device. Both a type and a number may be specified to select connectors when more than one connector of the same type exists on a device.

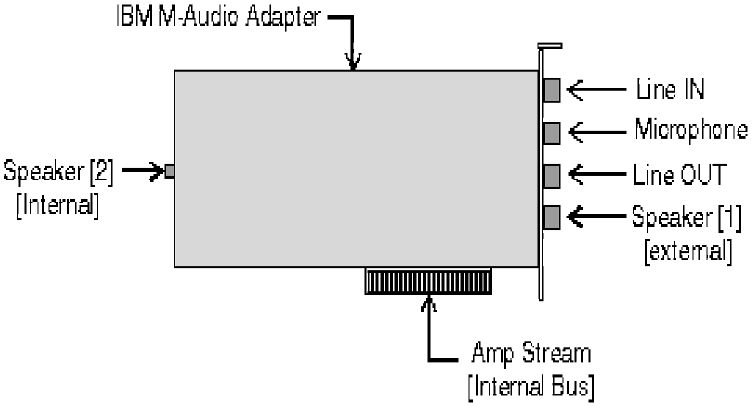
The following table describes connector types and typical uses.

Connector Type	Name	Description
MCI_MIDI_STREAM_CONNECTOR	midi stream	Digital input or output for the sequencer device. This information is typically streamed to an amplifier-mixer logical device.
MCI_CD_STREAM_CONNECTOR	cd stream	Digital output from a CD-ROM drive capable of reading the CD-DA data directly off the disc. This information is typically streamed to an amplifier-mixer logical device.
MCI_XA_STREAM_CONNECTOR	xa stream	The flow of digital audio information from a CD-ROM/XA drive capable of streaming the ADPCM data internally. This information is typically streamed to an amplifier-mixer logical device.
MCI_WAVE_STREAM_CONNECTOR	wave stream	Digital input or output for the waveaudio device. This information is typically streamed to an amplifier-mixer logical device.
MCI_AMP_STREAM_CONNECTOR	amp stream	The flow of information to an amplifier-mixer device. Typically this information comes from another logical device and can include MIDI, and various kinds of digital audio.
MCI_HEADPHONES_CONNECTOR	headphones	The connector on the device labeled for, or typically used, to attach headphones to the device.
MCI_SPEAKERS_CONNECTOR	speakers	The connector on the device labeled for, or typically used, to attach speakers to the device.
MCI_MICROPHONE_CONNECTOR	microphone	The connector on the device labeled for, or typically used, to attach a microphone to the device.

MCI_LINE_IN_CONNECTOR	line in	The connector on the device labeled for, or typically used to provide line level input to the device.
MCI_LINE_OUT_CONNECTOR	line out	The connector on the device labeled for, or typically used, to provide line level output from the device.
MCI_VIDEO_IN_CONNECTOR	video in	The connector on the device labeled for, or typically used to provide video input to the device.
MCI_VIDEO_OUT_CONNECTOR	video out	The connector on the device labeled for, or typically used, to provide video output from the device.
MCI_PHONE_SET_CONNECTOR	phone set	The connector on the device labeled for, or typically used, to attach a phone set to the device.
MCI_PHONE_LINE_CONNECTOR	phone line	The connector on the device labeled for, or typically used, to attach an external phone line to the device.
MCI_AUDIO_IN_CONNECTOR	audio in	The connector on the device labeled for, or typically used, to provide audio input to the device.
MCI_AUDIO_OUT_CONNECTOR	audio out	The connector on the device labeled for, or typically used, to provide audio output from the device.
MCI_UNIVERSAL_CONNECTOR	universal	A connector on a device which does not fall into any of the other categories. This connector type may be used to access device dependent function. The manufacturer of the device should define the exact use of this connector.

A Connector Example Using the IBM M-Audio Adapter

The following figure illustrates how the capabilities of an audio card might be modeled as an OS/2 amplifier-mixer device using connectors. This example uses the IBM M-Audio Capture and Playback Adapter, however a model can easily be defined for any manufacturer's audio card. The number and type of connectors may vary.



In this particular model, the *speakers(1)* connector is the default *speakers* connector and represents the physical speaker jack on the

M-Audio card. The *speakers(2)* connector represents the internal connection to the PC speaker. The *amp stream* connector represents the flow of digital information to and from the audio card.

Establishing Connections between Devices

A *connection* is the establishment of a flow of information from one device connector to a compatible connector on another device. One example of a connection is the attachment of a speaker to an audio card with speaker wire. Typically, an application might enable the speaker connector on an audio card, causing the flow of information out of the speaker connector. If you have connected the wire, then the audio is heard. In this case the multimedia system must rely on a person to make the actual connection.

Another category of connection exists where the connection is made internally in the computer, typically through the transfer of digital information from one media device to another. For instance, the waveaudio media device has a connection to its associated amplifier mixer device.

Default and Device Context Connections

Device connections are usually automatically established by the media device when the device is opened. The choice of connection is determined by a default established during installation of the media driver, and can be re-established using the MCI_DEFAULT_CONNECTION message.

Once opened, the media device may open and connect to another media device to provide the complete function of the originally opened device to the application. This is transparent to the calling application. One example is the waveaudio device, which uses a connected amplifier-mixer device to actually produce sound from the digital audio stream. The waveaudio device also uses the services of the amplifier-mixer device to set the volume.

While some services of the connected device can be surfaced in the definition of the originally opened device, the connected device can also provide some extended features beyond those required by the original device. If the application wishes to access these extended features, it can get the handle to the particular device context or *instance* of the connected device, using the MCI_CONNECTION message.

Note the subtle difference between a default connection and a device context connection. A default connection is the name of a connected device, whereas a device context connection is the actual handle to a particular instance of an opened device. An example of this is a waveaudio01 device that has a default connection to an ampmix01 device. When the waveaudio01 device is opened, it automatically opens the ampmix01 device, creating an instance of each device. Because devices may be shared in OS/2 multimedia, the waveaudio01 device can be opened again by another application and two new instances will be created. Although the default connection is the same in both cases, the device context connections are different.

Connectors Supported by Media Drivers

Each implementation of a media device defines the connectors that it supports. This information is maintained in the media driver and can vary with the underlying hardware.

The following table lists each media device and the connector types applicable to each device. The actual number and types of connectors in each device can vary from one implementation to another.

Device Type	Connectors
ampmix	amp stream, headphones, speakers, microphone, line in, line out
cdaudio	cd stream, headphones
cdxa	xa stream

digitalvideo	headphones, speakers, microphone, line in, line out, audio in, audio out
headphones	audio in
microphone	audio out
sequencer	midi stream, headphones, speakers, line out
videodisc	video out, line out
speakers	audio in
waveaudio	wave stream, headphones, speakers, microphone, line in, line out

---

Allowable Connections for Connector Types

For a connection to exist between two media devices, the connectors on each device must be of compatible types. For instance, a connection could be established between the line OUT connector on one device and the line IN connector on another device, however a connection between line OUT and video IN is prohibited. This mechanism will eliminate the majority of incorrect connections. The following table lists the allowable connections based on connector types. These connections are allowed in either direction. Please note that this table should be read as meaning that a device which has a connector of the type in the first column can connect to a device which has a connector of the type in the second column.

amp stream	<->	wave stream midi stream cd stream xa stream
line in	<->	line out audio out
audio in	<->	headphones speakers line out
video in	<->	video out
headphones	<->	audio in line in
speakers	<->	audio in line in
microphone	<->	audio out
phone line	<->	phone set

The connections in the table shown above are based on connector types, not device types. The list of connections might appear incorrect if the connector types are misinterpreted as device types. For example, "headphones <-> audio in" might be misinterpreted as meaning that headphones can be connected to an audio input. Referring to however, we see that the HEADPHONE device has a connector of type *audio in* . This connector is analogous to the plug on the headphones and, from the perspective of the headphones, it is input. We also see that the AMPMIX and CDAUDIO devices have connectors of type *headphones* . The connection is correct, because headphones can be connected to either the CD player (CDAUDIO) or to the audio adapter (AMPMIX).

Amplifier-Mixer Device

The OS/2 amplifier-mixer (ampmix) device is similar to a home stereo amplifier-mixer. Components are plugged into the amplifier-mixer so that audio signals can be transferred to a pair of attached speakers, headphones, or perhaps another device. A comparable example of connecting to another device is playing an old phonograph record, and recording the sound on a new DAT (Digital Audio Tape) deck. The ampmix is the center of all audio signals and provides input or output switching and sound shaping services such as volume, treble, or bass control.

The logical ampmix device in OS/2 supports both analog and digital devices. Other OS/2 multimedia logical devices may be connected to the ampmix device. Similar to the previous example, the CD audio logical device could provide an analog input to the ampmix device, which could then be recorded by the digital waveform audio device.

Both a logical ampmix device and the audio adapter performs all the functions surfaced by the ampmix device. Two important points are the *speaker* and *amp stream* connectors.

Although there is actually no visible speaker jack on the back of the audio card, it is a convenient fiction for an application to view the PC internal speaker as another set of speakers that might be plugged into the back of the audio card. Using the previously defined concept of a *connector* , an application can view all flows of information into and out of the ampmix device in a similar fashion. Selecting the internal speaker as opposed to the external speakers may require the ampmix device to issue a completely different set of instructions to the actual hardware device. The application, however, remains completely device independent.

The other features of the ampmix device are provided by either issuing commands to the hardware device, or emulating in software. One example of software emulation is the support of changing the volume over a period of time, or *fade in/fade out* . The audio card may only support setting the volume to a particular value, however the ampmix device can send a series of values to achieve the fade effect.

The Amp Stream Connector

The *amp stream* connector represents the flow of digital information to and from the ampmix device. Again similar to home stereo amplifier-mixers, the ampmix device by itself is not especially notable until another device is attached. Information is transferred from the device and played back on a pair of attached speakers. Note that the ampmix device is a conduit of information, and relies on another device to provide the flow of information. Therefore, commands for the transport of information (such as play, seek, or stop), are sent to the attached device. Commands for transforming the information (such as treble or bass) are sent directly to the ampmix device.

As a nicety for applications, the attached device will provide volume control, so that the application need not provide ampmix functions unless some advanced audio functions are required. The volume command is transparently routed to the attached ampmix device. If the application needs to talk directly to the ampmix device, the value of the stream connector may be queried using the MCI_CONNECTION message, which returns a device context connection. If the string interface is being used, an alias can be established for the connected device. Ampmix commands may then be sent directly to the ampmix device.

Some devices also provide a *connector service* , which also alleviates the need to talk directly to the ampmix device for frequently requested function. An example of this is the waveaudio device, which attempts to process requests for *speakers* and several other connector types. If the service is available from the associated ampmix device it is routed; otherwise, the function fails. The connectors and connector services provided by each OS/2 multimedia logical device are discussed in the section for that device.

Sharing the Amplifier-Mixer Device

Because many components of OS/2 multimedia utilize the amplifier-mixer device, it is typically opened *shareable* so that several devices can use the ampmix device simultaneously, or serially, in an application-window-focus driven sharing scheme. The Media Device Manager (MDM) is responsible for allocating the resources of the underlying hardware correctly and informs an application with the MM_MCIPASSDEVICE message whenever use of the ampmix device is gained or lost.

When other media devices in an application use the ampmix device, the amplifier-mixer becomes a source of contention, depending on the capabilities of the underlying audio adapter. For example, the IBM M-Audio adapter supports the simultaneous playback of two mono 22 kHz PCM waveforms. However, if a third waveform is started, one of the previous two waveforms must be suspended. The application that opened the waveform audio (waveaudio) device receives a MM_MCIPASSDEVICE message with an event of MCI_LOSING_USE. Following completion of the third waveform, the second waveform is automatically restored and can then play to completion. See Device Sharing By Applications for information on device sharing.

The OS/2 multimedia system manages all device sharing, and informs the application when the device is temporarily unavailable.

Audio Attributes

The OS/2 ampmix device provides the following control of audio signals. Current values of audio attributes are retrieved using the MCI_STATUS message. New values are set using the MCI_SET command.

Support of these features can vary by manufacturer. Some companies may develop ampmix devices for use in OS/2 multimedia that provide additional capabilities. To determine whether the device supports a feature, use the MCI_GETDEVCAPS message. If the feature is not supported, MCIERR_UNSUPPORTED_FLAG is returned.

volume

Sets mixer-channel volume level as a percentage of the maximum achievable effect. Volume of the left and right channels for stereo signals can be set simultaneously using the keyword **all**, or independently using keywords **left** and **right**. The keyword **over** can be added to fade the volume in or out over a specified period of time.

treble

Sets treble as a percentage of the maximum achievable effect. The effect applies to the final output mix. Any specification of a channel is ignored.

bass

Sets bass as a percentage of the maximum achievable effect. The effect applies to the final output mix. Any specification of a channel is ignored.

balance

Sets balance. Zero is full left balance, 100 is full right balance. The effect applies to the final output mix. Any specification of a channel is ignored.

pitch

Sets pitch as a percentage of the maximum achievable effect. The effect applies to the final output mix. Any specification of a channel is ignored.

gain

Sets gain as a percentage of the maximum achievable effect for the currently selected input.

monitor

Sets monitor **on** or **off**. This feature controls whether or not the signal from an input device is heard when it is routed to another device for recording.

mute

Sets mute **on** or **off**.

loudness

Sets loudness as a percentage of the maximum achievable effect.

mid

Sets mid as a percentage of the maximum achievable effect.

reverb

Sets reverb as a percentage of the maximum achievable effect.

auto level control

Sets auto-level control (ALC) as a percentage of the maximum achievable effect.

chorus

Sets chorus as a percentage of the maximum achievable effect.

crossover

Sets crossover as a percentage of the maximum achievable effect.

custom1

Sets first custom effect as a percentage of the maximum achievable effect.

custom2

Sets second custom effect as a percentage of the maximum achievable effect.

custom3

Sets third custom effect as a percentage of the maximum achievable effect.

stereoenhance

Sets stereo enhancement as a percentage of the maximum achievable effect.

Amp Mixer Connectors

The following connectors are typically supported by ampmix devices:

MCI_AMP_STREAM_CONNECTOR
MCI_HEADPHONES_CONNECTOR
MCI_LINE_IN_CONNECTOR
MCI_LINE_OUT_CONNECTOR
MCI_MICROPHONE_CONNECTOR
MCI_SPEAKERS_CONNECTOR

The number and type of connectors supported by an audio device varies by manufacturer. To determine which connectors are supported, an application can issue the MCI_CONNECTORINFO message.

The ampmix device provides audio attribute control for individual connectors. An application can set an audio attribute for a connector with MCI_SET by specifying the connector in the *ulValue* field of the MCI_AMP_SET_PARMS structure. If *ulValue* contains MCI_AMP_STREAM_CONNECTOR, the setting affects the global output of the device.

An application can query the capabilities of a connector by using the MCI_GETDEVCAPS_EXTENDED message in combination with the MCI_MIXER_LINE flag in the *ulExtended* field of the MCI_AMP_GETDEVCAPS_PARMS structure. The *ulAttribute* field contains the audio attribute, and the *ulValue* field contains the connector whose capabilities you are querying.

The following example illustrates how an application can determine whether it can set the volume for a particular connector.

```
ULONG                       rc;            /* Return code */
MCI_AMP_GETDEVCAPS_PARMS   mciAmpCaps;    /* Ampmix caps */
USHORT                      usDeviceID;    /* Device ID   */

/* Test mixer support for volume changes on the microphone */

mciAmpCaps.ulValue     = MCI_MICROPHONE_CONNECTOR;
mciAmpCaps.ulAttribute = MCI_AMP_CAN_SET_VOLUME;
mciAmpCaps.ulExtended  = MCI_MIXER_LINE;

rc = mciSendCommand(usDeviceID,
                    MCI_GETDEVCAPS,
                    MCI_WAIT|MCI_GETDEVCAPS_EXTENDED,
                    (ULONG) &mciAmpCaps,
                    0);
```

Synchronizing Audio Attribute Settings

In an organized multimedia environment applications must be able to synchronize their audio settings with other applications. Whenever a user makes changes to an application's audio settings, other applications need to be informed of the changes so they can update their settings to conform to the changes made by the user.

An application can request notification about mixer events by sending an MCI_MIXNOTIFY message with MCI_MIXNOTIFY_ON specified. The application will then receive an MM_MCIEVENT message whenever a mixer attribute is changed, or a connector is enabled or disabled. When the system passes the MM_MCIEVENT message, the *usEventCode* field of the *MsgParam1* parameter contains MM_MIXEVENT, and the *MsgParam2* parameter contains a pointer to MCI_MIXEVENT_PARMS.

Attention: To avoid creating a terminal loop, the application must not set an audio attribute while processing the MM_MCIEVENT message.

```
typedef struct_MCI_MIXEVENT_PARMS {

  ULONG   ulLength;         /* Length of structure             */
  HWND    hwndMixer;        /* Window to receive mixer changes */
  ULONG   ulFlags;          /* Either MCI_MIX_ATTRIBUTE
                               or MCI_MIX_CONNECTOR            */
  USHORT  usDeviceID;       /* Device ID to notify of change   */
  ULONG   ulDeviceType;     /* Device type that caused change  */
  ULONG   ulDeviceOrdinal;  /* Ordinal of device type          */
  ULONG   ulAttribute;      /* Attribute that changed          */
  ULONG   ulValue;          /* New value of changed attribute  */
  ULONG   ulConnectorType;  /* Connector type                  */
  ULONG   ulConnectorIndex; /* Connector index                 */
  ULONG   ulConnStatus;     /* Connector enabled/disabled      */

} MCI_MIXEVENT_PARMS;

typedef MCI_MIXEVENT_PARMS *PMCI_MIXEVENT_PARMS;
```

The *ulFlags* field contains one of the following values:

- MIX_ATTRIBUTE
- MIX_CONNECTOR

If the mixer event is the changing of an attribute, *ulAttribute* , *ulDeviceType* , and *ulValue* fields are valid.

If the mixer event is the changing of a connector, *ulConnectorType* , *ulConnectorIndex* , and *ulConnStatus* fields are valid.

The following example illustrates how an application can set up notification for every audio attribute change.

```
MCI_GENERIC_PARMS mixevent;

mixevent.hwndCallback = hwndmixer;

if (hMixer)
   {
   mciSendCommand(hMixer,
                  MCI_MIXNOTIFY,
                  MCI_WAIT | MCI_MIXNOTIFY_ON
                  (PVOID) &mixevent,
                  0);
```

When MCI_MIXNOTIFY_ON is specified, the *hwndCallback* field of MCI_GENERIC_PARMS must contain a valid window handle.

Direct Audio RouTines (DART) Interface

OS/2 multimedia provides application developers with a number of choices for playing and recording audio files. They can:

- Open the waveaudio device
- Open a memory playlist and use operation codes
- Use the direct audio interface of the ampmix device

The waveaudio device provides an easy-to-use interface that works well for applications with simple audio requirements. However time-critical game applications usually demand a faster response time than the waveaudio device can achieve, because of the device-independent layers of software between it and the audio device.

The memory playlist method of audio playback and recording reduces some of the system overhead that the waveaudio device incurs, because the playlist can stream audio data directly from application memory buffers to audio device buffers. But the playlist is not the ideal solution when precise buffer flow control is needed.

The Direct Audio RouTines (DART) interface for the ampmix device enables games and multimedia applications to bypass the waveaudio device entirely and communicate directly with the amp mixer. Using this interface, applications get the high-speed audio response they require, while remaining compatible with existing OS/2 multimedia applications. Because DART uses the media control interface (MCI), applications using DART can share the audio device with other applications simply by processing the MM_MCIPASSDEVICE message.

DART offers the following advantages:

- Preallocation of user memory
- No threads
- Function pointers instead of API
- Time-critical notifications
- Tailorable number of buffers

Using the DART Interface

To read and write audio data directly to the mixer device using the DART interface, the application does the following:

1. Opens the mixer device with MCI_OPEN
2. Initializes the mixer to use DART with MCI_MIX_SETUP
3. Allocates memory buffers with MCI_BUFFER
4. Uses function pointers to read and write data from the audio device
5. Uses MCI_STATUS_POSITION for precision timing
6. Controls data transfer with MCI_PAUSE, MCI_RESUME, MCI_STOP
7. Deallocates memory buffers with MCI_BUFFER
8. Closes the mixer device with MCI_CLOSE

Setting Up the Mixer

The application sends the MCI_MIX_SETUP message to the amp mixer to initialize the device for direct reading and writing of audio data in the correct mode and format-for example, PCM, MIDI, or MPEG audio.

If waveform audio data will be played or recorded, the application fills in the *ulDeviceType* field with MCI_DEVICETYPE_WAVEFORM_AUDIO. It must also provide values for the following digital-audio-specific fields: format tag, bits per sample, number of samples per second, and number of channels.

If MIDI data will be played, the application fills in the *ulDeviceType* field with a value of MCI_DEVICETYPE_SEQUENCER and places zeroes in the format specifications fields.

```
typedef struct_MCI_MIXSETUP_PARMS
  {
  HWND        hwndCallback;    /* IN  Window for notifications */
  ULONG       ulBitsPerSample; /* IN  Number of bits per sample */
  ULONG       ulFormatTag;     /* IN  Format tag */
  ULONG       ulSamplesPerSec  /* IN  Sampling rate */
  ULONG       ulChannels;      /* IN  Number of channels */
  ULONG       ulFormatMode;    /* IN  MCI_RECORD or MCI_PLAY */
  ULONG       ulDeviceType;    /* IN  MCI_DEVTYPE */
  ULONG       ulMixHandle;     /* OUT Read/Write handle */
  PMIXERPROC  pmixWrite;       /* OUT Write routine entry point */
  PMIXERPROC  pmixRead;        /* OUT Read routine entry point */
  PMIXEREVENT pmixEvent;       /* IN  Event routine entry point */
  PVOID       pExtendedInfo;   /* IN  Media-specific info */
  ULONG       ulBufferSize;    /* OUT Recommended buffer size */
  ULONG       ulNumBuffers;    /* OUT Recommended num buffers */
  } MCI_MIXSETUP_PARMS;
```

The application must also fill in the *pmixEvent* field of the MCI_MIXSETUP_PARMS structure with a function pointer for the mixer to use for event notification, such as a full buffer, empty buffer, or an error condition. If the call to the mixer is successful, it returns two function pointers to the application-one for reading data (mixRead) and the other for writing data (mixWrite) to the audio device.

An application can use the MCI_MIXSETUP_QUERYMODE flag to query a device to see if a particular mode is supported. The following example illustrates using MCI_MIXSETUP to query and prepare the audio device for playing 16-bit, 22050 KHz stereo mode.

```
   // MCI_MIXSETUP informs the mixer device of the entry point
   // to report buffers being read or written.
   // We will also need to tell the mixer which media type
   // we will be streaming.  In this case, we'll use
   // MCI_DEVTYPE_WAVEFORM_AUDIO.

    memset( &MixSetupParms, '\0', sizeof( MCI_MIXSETUP_PARMS ) );

   MixSetupParms.ulBitsPerSample = 16;
   MixSetupParms.ulFormatTag     = MCI_WAVE_FORMAT_PCM;
   MixSetupParms.ulSamplesPerSec = 22050;
   MixSetupParms.ulChannels = 2;    /* Stereo */
   MixSetupParms.ulFormatMode = MCI_PLAY;
   MixSetupParms.ulDeviceType = MCI_DEVTYPE_WAVEFORM_AUDIO;


    rc = mciSendCommand( usDeviceID,
                         MCI_MIXSETUP,
                         MCI_WAIT | MCI_MIXSETUP_QUERYMODE,
                         ( PVOID ) &MixSetupParms,
                         0 );


    if ( ULONG_LOWD( rc ) != MCIERR_SUCCESS )
           {
           CHAR  szError[255];
             // The device can't handle this format
             // get an English error message for the caller.
           mciGetErrorString( ULONG_LOWD( rc ), szError, 255 );
           printf("Can't play because of %s", szError );
           exit( 1 );
           }

   // The mixer will inform us of entry points to
   // read/write buffers to and also give us a
   // handle to use with these entry points.

   MixSetupParms.pmixEvent = MyEvent;


    rc = mciSendCommand( usDeviceID,
                         MCI_MIXSETUP,
                         MCI_WAIT | MCI_MIXSETUP_INIT,
                         ( PVOID ) &MixSetupParms,
                         0 );
```

Allocating Memory Buffers

After the mixer device is set up to use DART, the application instructs the device to allocate memory by sending the MCI_BUFFER message with the MCI_ALLOCATE_MEMORY flag set. The application uses the MCI_BUFFER_PARMS structure to specify the number of buffers it wants and the size to be used for each buffer.

Note: Because of device driver restrictions, buffers are limited to 64KB on Intel-based systems. No such limit exists on PowerPC systems.

The *pBufList* field contains a pointer to an array of MCI_MIX_BUFFER structures where the allocated information is to be returned.

```
typedef struct_MCI_BUFFER_PARMS {
  HWND   hwndCallback;    /* Window for notifications */
  ULONG  ulStructLength;  /* Length of MCI_BUFFER_PARMS */
  ULONG  ulNumBuffers;    /* Number of buffers to allocate (IN/OUT)*/
  ULONG  ulBufferSize;    /* Size of buffers mixer should use */
  ULONG  ulMintoStart;    /* Unused */
  ULONG  ulSrcStart;      /* Unused */
  ULONG  ulTgtStart;      /* Unused */
  PVOID  pBufList;        /* Pointer to array of buffers */
} MCI_BUFFER_PARMS;

typedef MCI_BUFFER_PARMS *PMCI_BUFFER_PARMS;
```

The following example illustrates using MCI_BUFFER to allocate memory.

```
   MCI_MIX_BUFFER   MyBuffers[ MAX_BUFFERS ];

    BufferParms.ulNumBuffers = 40;
    BufferParms.ulBufferSize = 4096;
    BufferParms.pBufList = MyBuffers;

     rc = mciSendCommand( usDeviceID,
                  MCI_BUFFER,
                  MCI_WAIT | MCI_ALLOCATE_MEMORY,
                  ( PVOID ) &BufferParms,
                  0 );

     if ( ULONG_LOWD( rc) != MCIERR_SUCCESS )
        {
        printf( "Error allocating memory.  rc is : %d", rc );
        exit ( 1 );
        }

     // MCI driver will return the number of buffers it
     // was able to allocate
     // it will also return the size of the information
     // allocated with each buffer.

     ulNumBuffers = BufferParms.ulNumBuffers;

     for ( ulLoop = 0; ulLoop < ulNumBuffers; ulLoop++ )
        {
        rc = mmioRead ( hmmio,
                      MyBuffers[ ulLoop ].pBuffer,
                      MyBuffers[ ulLoop ].ulBufferLength);

        if ( !rc )
           {
           exit( rc );
           }
        MyBuffers[ ulLoop ].ulUserParm = ulLoop;

        }
```

Reading and Writing Data

The MCI_MIX_BUFFER structure is used for reading and writing data to and from the mixer.

Once the device is set up and memory has been allocated, the application can use the function pointers obtained during MCI_MIXSETUP to communicate with the mixer. During a playback operation, the application fills the buffers with audio data and then writes the buffers to the mixer device using the *pmixWrite* entry point. When audio data is being recorded, the mixer device fills the buffers using the *pmixRead* entry point. Each buffer returned the the application has a time stamp (in milliseconds) attached so the program can determine the current time of the device.

```
typedef struct_MCI_MIX_BUFFER {
  ULONG  ulStructLength;  /* Length of the structure */
  ULONG  pBuffer;         /* Pointer to a buffer */
  ULONG  ulBufferLength;  /* Length of the buffer */
  ULONG  ulFlags;         /* Flags */
  ULONG  ulUserParm;      /* User buffer parameter */
  ULONG  ulTime;          /* Device time in milliseconds */
  ULONG  ulReserved1;     /* Unused */
  ULONG  ulReserved2;     /* Unused */
  } MCI_MIX_BUFFER;
typedef MCI_MIX_BUFFER *PMCI_MIX_BUFFER
```

MCI_STOP, MCI_PAUSE, and MCI_RESUME are used to stop, pause, or resume the audio device, respectively. MCI_STOP and MCI_PAUSE can only be sent to the mixer device after mixRead and mixWrite have been called. MCI_RESUME will only work after MCI_PAUSE has been sent.

Note: After your application has completed data transfers, issue MCI_STOP to avoid a pause the next time the mixer device is started.

If your application needs more precise timing information than provided by the time stamp returned with each buffer, you can use MCI_STATUS with the MCI_STATUS_POSITION flag to retrieve the current time of the device in MMTIME units.

Master Volume and the Ampmix Device

The maximum volume level of all logical devices in the system are controlled by the Volume Control application supplied with OS/2 multimedia. The Volume Control application sets a scale by which all subsequent volume commands to the ampmix device are based. For instance, if the Volume Control sets the master volume at 50%, then all volume levels are cut in half.

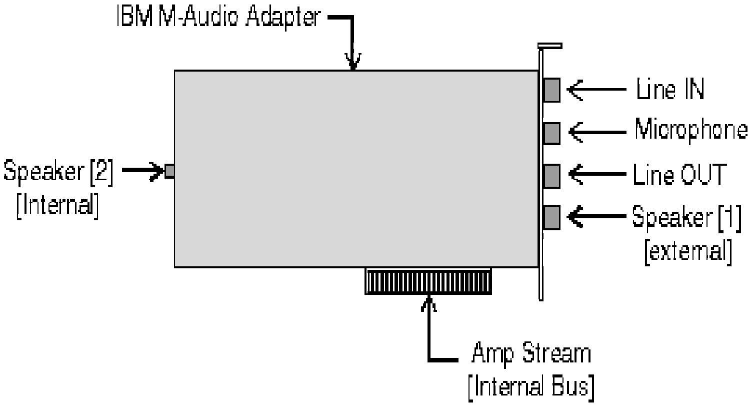
Some devices may only support two levels of volume (on/off). These devices are off when the master volume is set to zero, and on at any other value.

Note: While the MCI_MASTERAUDIO message can be sent by any application, only the master volume application or a replacement should utilize this message to set the master volume. Master volume should only be controlled at the discretion of an end user as implemented in the Volume Control application. A parameter of the MCI_MASTERAUDIO message allows an application or a media driver to query the master volume level.

Amplifier-Mixer Command Messages

Message	Description
MCI_BUFFER	Allocates or deallocates memory for use with the audio device.
MCI_CONNECTOR	Enables, disables, or queries the status of a connector on a device.
MCI_CLOSE	Closes the amp mixer instance.
MCI_GETDEVCAPS	Gets device capabilities.
MCI_INFO	Gets device information.
MCI_MIX_SETUP	Sets up the device in the correct mode (for example, PCM, MPEG audio, or MIDI).
MCI_OPEN	Opens an instance of the amp mixer.
MCI_PAUSE	Pauses playback or recording.
MCI_RESUME	Resumes playback or recording.
MCI_SET	Sets audio attributes using the MCI_AMP_SET_PARMS structure. For a list of supported attributes, see Audio Attributes.
MCI_STATUS	Gets device status.
MCI_STOP	Stops playback or recording.

M-Audio Adapter Specifics



1. The speakers (1) connector is the external speakers jack on the back of the card. The speakers (2) connector is really the internal PC speaker.

2. The Line OUT and speakers (1) connectors can be enabled or disabled by the ampmix device, although the adapter is incapable of actually switching the output. The ampmix device does report that the connector is actually enabled or disabled.

3. The speakers(2) connector can be enabled or disabled, resulting in the PC internal speaker being turned on or off.

4. The microphone and line IN connectors are mutually exclusive. Enabling one connector automatically disables the other. Disabling both connectors automatically enables the microphone.

5. The amp stream connector represents the transfer of digital audio information to and from the M-Audio card. This connector is always enabled.

6. The M-Audio adapter does not support independent control of volume for the left and right channels of a stereo signal. Any device connected to an M-Audio amplifier-mixer device returns MCIERR_UNSUPPORTED_FLAG if an attempt is made to independently control the volume of the left and right channels with the MCI_SET command.

Waveform Audio Device

The OS/2 waveform audio (waveaudio) device allows an application to play or record digital audio using files or application memory buffers. While *audio* refers to the sound waves (changes in air pressure) that have a perceived effect on the human ear, *waveform* refers to a digital representation of the original audio sound wave. Using one technique called pulse code modulation (PCM), discrete samples of the sound wave are encoded by an audio adapter at precise intervals. The numerical value of the sample increases when the sound wave's force (loudness) increases. The variation of the sample increases as the frequency of the sound wave increases.

The number of samples per second taken of the original sound wave as well as the precision (or resolution) of the sample dictate the quality of the sound reproduction. Typical sampling rates include 44 kHz, 22 kHz, and 11 kHz, where kHz is an abbreviation for kilohertz or thousands of cycles per second. The sampling precision is usually measured in bits where 8 or 16 bits per sample are representative of most audio adapters. Mono or stereo refers to the number of channels transferring digital audio. *Mono* represents one channel and *stereo* represents two channels.

Generally, the higher the sampling rate and resolution, the higher the perceived quality; however this comes at the expense of potentially enormous data rates and file sizes. For example, audio quality equivalent to that produced by a CD audio device requires a sampling rate of 44.1 kHz, and 16-bit resolution for each of the channels in a stereo recording. This information alone results in a data rate of 176 kilobytes per second! Luckily, many applications of digital audio are adequately supported with sampling rates and resolutions as low as 22 kHz and 8 bits respectively. The exact choice of parameters will vary, depending on the requirements of the application.

The Wave Stream Connector

The *wave stream* connector represents the flow of digital information to and from the waveaudio device to its associated amplifier-mixer (ampmix) device. During playback, the waveaudio device sends digitized sounds from either application memory or files to the ampmix device for subsequent conversion into audio that can be heard through conventional speakers or headphones. When recording, the waveaudio device receives waveforms from the ampmix device and stores the digital information in a file or in application memory.

Control of the characteristics of the waveform information is provided by the waveaudio device. The quality of the waveform can be controlled by setting the format, sampling rate, bits per sample, and the number of channels. As an additional service, the waveaudio device will also allow the volume to be controlled. This service is actually provided by the ampmix device in a way that is transparent to the calling application. If other advanced audio shaping features are required, the application can retrieve the device ID of the ampmix device using the MCI_CONNECTION message. Once the device ID has been obtained, the application can send commands directly to the ampmix device. Examples include **set** commands to manipulate treble, bass, and balance.

Waveaudio Device Features

- Multiple time formats
- Waveform characteristics

  Data Format
  File format (RIFF WAVE, AVC, or others if an MMIO procedure is supplied)
  Sampling rate
  Bits per sample (resolution)
  Number of channels

- Playback and record sources

  File system
  Application memory

- Audio shaping

  Volume control
  Other features that might be available through the associated ampmix device

- Cue point and position advise notification

Waveform Data Formats

There are several formats used for storing waveform data within a computer system. OS/2 multimedia recognizes several resolutions of the Pulse Code Modulation (PCM) format, because it is supported by most audio adapters. OS/2 multimedia also recognizes ADPCM formats. Refer to the Appendix of the *OS/2 Multimedia Programming Reference* for descriptions of these formats.

Pulse Code Modulation (PCM) refers to the variation of a digital signal to represent audio amplitude. This method of assigning binary values to amplitude levels supports the conversion of analog signals to digital signals by adapters such as the M-Audio Capture and Playback Adapter.

Adaptive Differential Pulse Code Modulation (ADPCM) is a technique for compressing waveform samples. ADPCM can reduce the amount of data storage required by a factor of 16 to 1, but some price is paid in fidelity for the higher compression rates.

M-Audio Adapter Specifics

The following tables list the valid MCI_WAVE_SET items for the operation modes currently supported by the M-Audio Capture and Playback Adapter.

Note: Numbers in table stand for number of channels supported in this mode-- mono (1), stereo (2), N/A (Not Available)

Data Size	8000 Hz	11025 Hz	22050 Hz	44100 Hz
8-bit	1, 2	1, 2	1, 2	1, 2
16-bit	1, 2	1, 2	1, 2	1, 2

Data Size	8000 Hz	11025 Hz	22050 Hz	44100 Hz
8-bit	N/A	N/A	N/A	N/A
16-bit	N/A	1	1, 2	1

Data Size	8000 Hz	11025 Hz	22050 Hz	44100 Hz
8-bit	1,2	1,2	1,2	1,2
16-bit	N/A	N/A	N/A	N/A

Data Size	8000 Hz	11025 Hz	22050 Hz	44100 Hz
8-bit	1,2	1,2	1,2	1,2
16-bit	N/A	N/A	N/A	N/A

Audio Device Capabilities

If MCI_GETDEVCAPS_EXTENDED is specified in conjunction with MCI_GETDEVCAPS_ITEM, the MCI_GETDEVCAPS_WAVE_FORMAT value can be placed in the *ulItem* field for the waveaudio device as an extended request. The MCI_GETDEVCAPS_WAVE_FORMAT value allows an application to query if the device supports a specific waveaudio format. The application must fill in the *ulBitsPerSample* , *ulFormatTag* , *ulSamplesPerSec* , *ulChannels* , and *ulFormatMode* fields in the MCI_WAVE_GETDEVCAPS_PARMS data structure. The driver returns MCI_TRUE if the format is supported or returns a specific error describing why the command field failed if the format is not supported.

The following code fragment shows a portion of the Audio Recorder Sample program provided in the Toolkit (\TOOLKIT\SAMPLES\MM\RECORDER). This program uses the MCI_GETDEVCAPS message to determine the capabilities of the currently selected waveaudio device.

```
ULONG                      ulRC;         /* return code from function */
MCI_WAVE_GETDEVCAPS_PARMS mciAudioCaps; /* MCI_GETDEVCAPS_PARMS
                                           structure */
memset( &mciAudioCaps , 0, sizeof(MCI_WAVE_GETDEVCAPS_PARMS));
```

```
  /* Test to see if the device can play 11 kHz, 8-bit, mono files. */
  mciAudioCaps.ulBitsPerSample = 8;
  mciAudioCaps.ulFormatTag     = DATATYPE_WAVEFORM;
  mciAudioCaps.ulSamplesPerSec = 11025;
  mciAudioCaps.ulChannels      = 1;
  mciAudioCaps.ulFormatMode    = MCI_PLAY;
  mciAudioCaps.ulItem          = MCI_GETDEVCAPS_WAVE_FORMAT;

  ulRC = mciSendCommand (mciOpenParms.usDeviceID,   /* Device ID    */
                         MCI_GETDEVCAPS,
                         MCI_WAIT | MCI_GETDEVCAPS_EXTENDED
                              | MCI_GETDEVCAPS_ITEM,
                         (PVOID) &mciAudioCaps,
                         0);
       .
       .
       .

/* Test to see if the device can record 11 kHz, 16-bit, mono files. */
  mciAudioCaps.ulBitsPerSample = 16;
  mciAudioCaps.ulFormatTag     = DATATYPE_WAVEFORM;
  mciAudioCaps.ulSamplesPerSec = 11025;
  mciAudioCaps.ulChannels      = 1;
  mciAudioCaps.ulFormatMode    = MCI_RECORD;
  mciAudioCaps.ulItem          = MCI_GETDEVCAPS_WAVE_FORMAT;

  ulRC = mciSendCommand (mciOpenParms.usDeviceID,  /* Device ID     */
                         MCI_GETDEVCAPS,
                         MCI_WAIT | MCI_GETDEVCAPS_EXTENDED
                              | MCI_GETDEVCAPS_ITEM,
                         (PVOID) &mciAudioCaps,
                         0);
```

Using the Waveform Audio Device

Because the waveaudio device is a compound device, it requires a device element. The device element is typically a file that contains a sampled waveform for playback. The waveaudio device can be opened with or without a device element. A device element can subsequently be specified using the **load** command.

Opening the Waveform Audio Device

The following string commands open the default waveaudio device and load a file onto it.

```
open waveaudio alias wave shareable
load wave c:\mysounds\train.wav
```

OS/2 multimedia allows you to specify the device to be used for a particular file based on the file's extension or its extended attributes (EAs). Using .TYPE EAs is the preferred method, because they remain with the files even when the files are renamed. Both file extensions and extended attributes can be associated with a device using the Multimedia Setup application. For instance, assuming files with an extension of .WAV have been associated with the waveaudio device, the following command will result in a file being loaded into the waveaudio device:

```
open c:\mysounds\monkey.wav alias monkey shareable
```

Finally, both the device element and the device type can be specified:

```
open c:\mysounds\paperjam.wav type waveaudio alias wave shareable
```

Recording a Waveform File

One of the typical uses of the waveform audio device is to digitize an input signal or sound into discrete samples for storage in a file. An example of this would be recording an electronic audio mail message to actually *tell* someone about an idea, as opposed to typing a memo on the same subject. An electronic audio mail application would be completely shielded from the complexity of digitizing a signal and would only need to specify a file, while providing the user with a simple control panel to allow the message to be recorded. The user might press a stop button on the control panel when finished describing the idea. The application could then issue a **stop** command to the waveaudio device to discontinue the recording.

```
open myidea.wav waveaudio alias wave wait
record wave notify
.
.
.
** recording the idea into myidea.wav **
.
.
.
stop wave wait
```

Like many text editors, the waveform audio media driver will not actually modify the original file until it receives a command to save the changes. Any temporary files created during the record operation will be located in the directory specified by the MSV_WORKPATH multimedia system variable. The path can be specified on the *system* page of the Multimedia Setup application. The use of temporary files is completely transparent to the application.

The file can be saved using the original file name, or a new file name can be specified. If a **save** command is not issued before closing the waveform audio device, all changes will be discarded.

```
save wave wait
close wave wait
```

It is possible to open or load the waveaudio device specifying a special *readonly* option. In this mode, the waveaudio device prevents any modification to the file from either the **save** or **record** commands. In certain circumstances, the driver might be able to optimize performance by utilizing the information that the file will not be modified. The option will also allow multiple applications to share the same file for playback purposes and will prevent inadvertent modification of the file.

```
open bigwave.wav type waveaudio alias wave readonly shareable
```

Creating New Files

The waveaudio device will create a new file on either the MCI_OPEN or MCI_LOAD commands if a file element is indicated (MCI_OPEN_ELEMENT_ID) and the specified file name does not exist. If no file name is indicated, the waveaudio driver will create an unnamed temporary file. If an unnamed temporary file is created, it can later be named by issuing the MCI_SAVE command, which must include the permanent name of the new file.

To support file creation from the string interface, a special file name called **new** is reserved for system use. This file name should be used in place of the usual application supplied file name. As in the command message interface, the **save** command must be issued to give the file a permanent name.

```
open new type waveaudio alias wave wait
record wave notify
.
.
```

```
.
** recording **
.
.
.
stop wave wait
save wave myspeech.wav wait
```

When a file is initially created, default settings will be assigned by the media driver and might depend on the capabilities of the audio adapter. The IBM waveform audio driver will use PCM, 22 kHz, 16 bits per sample, and mono as the default for 16-bit adapters. If the adapter does not support 16-bit PCM, then the resolution (bits per sample) will be downgraded to 8 bits.

The following table lists audio adapters supported by OS/2 multimedia. The default settings are those initially assigned by the media driver to a new file when that particular audio adapter is being used.

Audio Adapter	Format	Sampling Rate	Bits per Sample	Channels
IBM M-Audio	PCM	22 kHz	16	1
Sound Blaster	PCM	22 kHz	8	1
Sound Blaster Pro	PCM	22 kHz	8	2
Sound Blaster 16	PCM	22 kHz	16	1
Pro AudioSpectrum 16	PCM	22 kHz	16	1

OS/2 multimedia enables recording of digital audio information in the format that fits specific needs, such as space or quality. For example, assume that a new waveaudio file is created with the following command:

```
open new type waveaudio alias a wait
```

When the file is created, you might want a file that is compatible with mu-law (the compression scheme used by the telephone system). To change the compression scheme, the format tag must be set for the file. The following string commands prepare the file for recording mu-law by setting the format tag:

```
set a format tag mulaw wait
set a bitspersample 8 wait
set a channels 1 wait
set a samplespersec 11025 wait
```

If you wanted to record with a compression scheme commonly used in Europe (a-law), the following command could have been issued:

```
set a format tag alaw wait
```

An application should always set the waveform format, sampling rate, resolution, and number of channels to ensure that the waveform is created with the desired parameters as shown in the following string interface example.

```
set wave format tag PCM wait
set wave samplespersec 22050 wait
set wave bitspersample 8 wait
set wave channels 1 wait
```

Note: When modifying the settings on a waveaudio device, the *format tag* should be changed first, because it might force the automatic modification of other settings to make them compatible with the new format. For instance, a waveaudio device that supports 16-bit PCM might only support 8-bit ADPCM. Changing the format from PCM to ADPCM will automatically change the bits per sample setting.

Playing and Recording non-RIFF Waveforms

The waveform audio device will create new waveforms according to the RIFF WAVE data standard. It is possible, however to play other data formats using OS/2 multimedia if the appropriate MMIO procedure has been supplied. The selection of the appropriate I/O procedure (IOProc) is transparent to the application if the IOProc has been installed.

One example of this feature is OS/2 multimedia's ability to play waveform audio files that were created using IBM's AVC application and the M-Audio card. Note that the AVC support provides playback capabilities only. The waveform audio device will temporarily report FALSE to the save and record capabilities of the device capabilities (MCI_GETDEVCAPS) function when the underlying I/O procedure does not support the creation of files. Applications should check the device capabilities to appropriately display a user interface that reflects the true capabilities of the waveaudio driver and its associated I/O procedure.

For example, a waveform editor application should grey out its record button when an AVC file is loaded, as only playback operations are supported. Querying the device capabilities would return FALSE for **can record**. If a waveform file in the RIFF WAVE format is subsequently loaded, the record button should be enabled, because the same **can record** query will now return TRUE. In all instances, by using the high-level OS/2 multimedia mciSendString or mciSendCommand interface to reference device capabilities, the application is shielded from the underlying implementation.

Creating a Waveform Playlist

Specialized applications such as a waveform editor might require the capability of playing and recording using application memory buffers instead of files. The *memory playlist* feature of OS/2 multimedia provides the construct for supplying memory buffers to the waveaudio device. Besides implementing simple circular buffering schemes, memory playlists can be used to synthesize complex and unique waveform sounds. By following each DATA statement with a MESSAGE statement, an application can be informed as to when the buffer can be reused.

Playlist Structure

Depending on the complexity of the application, memory playlists can be used to provide a single large memory buffer, or multiple buffers in a circular buffering scheme. The following is an example of how a memory playlist might be constructed to implement a simple circular buffering scheme.

```
    0:    NOP
    1:    DATA...
    2:    MESSAGE...
    3:    DATA...
    4:    MESSAGE...
    5:    DATA...
    6:    MESSAGE...
    7:    BRANCH 0
```

Note that regardless of whether the playlist is being used for play or record operations, the MESSAGE instruction will notify the application when the playlist processor has consumed or filled the preceding DATA buffer. An MM_MCIPLAYLISTMESSAGE will be sent to the window procedure specified when the waveaudio device was originally opened.

The following code fragment shows the SetUpPlaylist procedure that is performed once, during initialization of the Clock Sample program. It calls the procedure CopyWaveformIntoMemory to copy the waveform files into memory buffers. It also initializes the playlist data structure by supplying the address and size of the memory buffers holding the data in the appropriate data structure fields.

```
VOID SetupPlayList( VOID )
{
 /*
  * This array keeps the address of each audio chime file.
  */
 static LONG *pulBaseAddress[ NUMBER_OF_CHIME_FILES ];

 USHORT usChimeFileId;            /* Chime audio file ID.        */
 ULONG  ulSizeOfFile,             /* Size of audio file.         */

ulMemoryAllocationFlags = PAG_COMMIT | PAG_READ | PAG_WRITE;
for(usChimeFileId=0; usChimeFileId<NUMBER_OF_CHIME_FILES;
    usChimeFileId++)
 {
    ulSizeOfFile = HowBigIsTheChimeFile( usChimeFileId );
  /*
   * If the returned file size is 0, there is a problem with the
   * chime files.  A message will already have been shown to the user
   * by the HowBigIsTheChimeFile function so exit this routine.
   */
    if ( ulSizeOfFile == 0 )
    {
       return;
    }
    if ( (pulBaseAddress[ usChimeFileId ] = (LONG *)
            malloc( ulSizeOfFile )) == (LONG *) NULL )
    {
       /*
        * The memory for the waveform files cannot be allocated.
        * Notify the user and return from this routine.  No playlist
        * can be created/played until memory is available.
        */
       ShowAMessage(
          acStringBuffer[
             IDS_NORMAL_ERROR_MESSAGE_BOX_TEXT - 1 ],
          IDS_CANNOT_GET_MEMORY, /* ID of the message to show. */
          MB_OK | MB_INFORMATION | MB_HELP |  MB_APPLMODAL |
             MB_MOVEABLE );           /* Style of the message box. */

       return;

    }  /* End of IF allocation fails. */
    /*
     * Place the waveform files into the memory buffer that was just
     * created.
     */
    CopyWaveformIntoMemory(
       pulBaseAddress[ usChimeFileId ],
       ulSizeOfFile,
       usChimeFileId );
    /*
     * Now that we've loaded the waveform into memory, we need to put
     * its address and size into the playlist data statements that
     * use this particular file.
     *
     * Its address must be placed into the data statement's first
     * operand and its size must be placed in the data
     * statement's second operand.
     *
     * For the four different playlists, one for each chime time
     * (1/4, 1/2, 3/4 and 1 hour increments),
     * the address of the chime file and its size will be loaded
     * into each data statement of the Playlist.
     */
    if ( usChimeFileId == 0 )
    /* If we just loaded CLOCK1.WAV */
    {
       /*
        * Put the address of this chime into the first operand of
        * every data operation that uses this particular chime.
        */
    apltPlayList[ 0 ][ 0 ].ulOperandOne =  /* 1/4 hour 1st data op */
    apltPlayList[ 1 ][ 0 ].ulOperandOne =  /* 1/2 hour 1st data op */
    apltPlayList[ 2 ][ 0 ].ulOperandOne =  /* 3/4 hour 1st data op */
    apltPlayList[ 2 ][ 2 ].ulOperandOne =  /* 3/4 hour 3rd data op */
    apltPlayList[ 3 ][ 0 ].ulOperandOne =  /* 1   hour 1st data op */
    apltPlayList[ 3 ][ 2 ].ulOperandOne =  /* 1   hour 3rd data op */
       (ULONG) pulBaseAddress[ usChimeFileId ];  /* address        */
    /*
     * Now put the size of the file into the second operand of every
     * data operation that uses this particular chime.
     */
    apltPlayList[ 0 ][ 0 ].ulOperandTwo =  /* 1/4 hour 1st data op */
    apltPlayList[ 1 ][ 0 ].ulOperandTwo =  /* 1/2 hour 1st data op */
    apltPlayList[ 2 ][ 0 ].ulOperandTwo =  /* 3/4 hour 1st data op */
    apltPlayList[ 2 ][ 2 ].ulOperandTwo =  /* 3/4 hour 3rd data op */
    apltPlayList[ 3 ][ 0 ].ulOperandTwo =  /* 1   hour 1st data op */
    apltPlayList[ 3 ][ 2 ].ulOperandTwo =  /* 1   hour 3rd data op */
       ulSizeOfFile;                             /* size         */
 }
 else
 if ( usChimeFileId == 1 )
```

```
 /* If we just loaded CLOCK2.WAV */
 {
    /*
     * Put the address of this chime into the first operand of
     * every data operation that uses this particular chime.
     */
    apltPlayList[ 1 ][ 1 ].ulOperandOne =  /* 1/2 hour 2nd data op */
    apltPlayList[ 2 ][ 1 ].ulOperandOne =  /* 3/4 hour 2nd data op */
    apltPlayList[ 3 ][ 1 ].ulOperandOne =  /* 1   hour 2nd data op */
    apltPlayList[ 3 ][ 3 ].ulOperandOne =  /* 1   hour 4th data op */
       (ULONG) pulBaseAddress[ usChimeFileId ];  /* address    */
    /*
     * Now put the size of the file into the second operand of every
     * data operation that uses this particular chime.
     */
    apltPlayList[ 1 ][ 1 ].ulOperandTwo =  /* 1/2 hour 2nd data op */
    apltPlayList[ 2 ][ 1 ].ulOperandTwo =  /* 3/4 hour 2nd data op */
    apltPlayList[ 3 ][ 1 ].ulOperandTwo =  /* 1   hour 2nd data op */
    apltPlayList[ 3 ][ 3 ].ulOperandTwo =  /* 1   hour 4th data op */
       ulSizeOfFile;                             /* size        */
 }
 else
 if ( usChimeFileId == 2 )
 /* If we just loaded CLOCK3.WAV         */
 /* (this is the gong part of the chime) */
{
    /*
     * Put the address of this chime into the first operand of
     * every data operation that uses this particular chime.
     */
    apltPlayList[ 3 ][ 5 ].ulOperandOne =  /* 1 hour 5th data op */
       (ULONG) pulBaseAddress[ usChimeFileId ];

    /*
     * Now put the size of the file into the second operand of every
     * data operation that uses this particular chime.
     */

    apltPlayList[ 3 ][ 5 ].ulOperandTwo =   /* 1 hour 5th data op */
       ulSizeOfFile;
  }

 }  /* End of For loop of chime files. */

}  /* End of SetupPlayList */
```

Suggested Setups for Playlists

The following tables provide guidelines for setting up playlists to utilize memory more efficiently.

For a sampling rate of 8 kHz:

Bits Per Sample	# of Channels	Buffer Size	Max. # of Buffers	# of Buffers before a Stream is Started
8	1	4KB	30	3
8	2	8KB	30	3
16	1	8KB	30	3
16	2	16KB	20	3

For a sampling rate of 11 kHz:

Bits Per Sample	# of Channels	Buffer Size	Max. # of Buffers	# of Buffers before a Stream is Started
8	1	4KB	40	3
8	2	8KB	40	3
16	1	8KB	40	3
16	2	16KB	30	3

For a sampling rate of 22 kHz:

Bits Per Sample	# of Channels	Buffer Size	Max. # of Buffers	# of Buffers before a Stream is Started
8	1	8KB	40	3
8	2	16KB	30	3
16	1	16KB	30	3
16	2	32KB	20	3

For a sampling rate of 44 kHz:

Bits Per Sample	# of Channels	Buffer Size	Max. # of Buffers	# of Buffers before a Stream is Started
8	1	16KB	30	3
8	2	32KB	20	3
16	1	32KB	20	3
16	2	60KB	10	5

Waveform Audio Command Messages

Following are descriptions of the command messages used to control the recording, editing, and playback of waveform data. Analog input devices for recording waveforms are a microphone and tape deck. Analog output devices for waveform playback are a stereo amplifier or speakers connected to an audio adapter.

Message	Description
MCI_CLOSE	Closes the waveform audio player.
MCI_CONNECTOR	Enables or disables a connector, queries its state, or identifies its type.
MCI_COPY	Copies data from the device element to the clipboard or buffer.

MCI_CUE	Cues the device for minimum delay in recording or playback.
MCI_CUT	Removes data from the device element and places it in the clipboard or buffer.
MCI_DELETE	Removes the specified range of data from the device element.
MCI_GETDEVCAPS	Gets device capabilities.
MCI_INFO	Gets the name of the currently loaded file.
MCI_LOAD	Loads a waveform data file.
MCI_OPEN	Initializes the waveform audio player.
MCI_PASTE	Issues a DELETE on the selected range and inserts data into clipboard or buffer.
MCI_PAUSE	Suspends the current play or record action.
MCI_PLAY	Plays back waveform data by means of an audio adapter:
MCI_RECORD	Records waveform data.
MCI_REDO	Redoes the CUT, DELETE, PASTE, or RECORD operation most recently done by MCI_UNDO.
MCI_RESUME	Resumes the current play or record action from a paused state.
MCI_SAVE	Saves the device element in its current format.
MCI_SEEK	Seeks to a specified location.
MCI_SET	Sets device information.
MCI_SET_CUEPOINT	Sets a cue point.
MCI_SET_POSITION_ADVISE	Sets a position change notification request.
MCI_SET_SYNC_OFFSET	Sets a synchronization offset.
MCI_STATUS	Receives status on current settings for items used for recording, playback, and saving.
MCI_STOP	Stops the waveform device before loading a new file.
MCI_UNDO	Undoes the operation most recently performed by CUT, DELETE, PASTE, or RECORD.

Waveaudio Connectors

The waveaudio device directly supports one *wave stream* connector which is always enabled. As it is likely that an application will need to select the recording source or the output destination on the amplifier-mixer device, the waveaudio device will attempt to provide the following connector services to an application. If the requested connector is not available, the command will fail.

- headphones
- speakers
- line out
- microphone

- line in

Additional connectors might be available on the ampmix device. To control these connectors obtain the device ID of the ampmix device using the MCI_CONNECTION message and issue the connector command directly to the associated amplifier-mixer.

To determine which connectors are supported by an amplifier-mixer device, use the MCI_CONNECTORINFO message.

The Audio Recorder Sample program illustrates the concept of recording audio data. In order to do this, it first configures the device settings such as the input source as shown in the following code fragment.

```
 MCI_CONNECTOR_PARMS  mciConnectorParms;   /* for MCI_CONNECTOR  */
/*
 * Set up input source - microphone or line in.
 * Initialize MCI_CONNECTOR_PARMS structure with the pertinent
 * information, and then issue an MCI_CONNECTOR command by way of
 * mciSendCommand.
 */
 mciConnectorParms.ulConnectorType = usDeviceType;
                                          /* microphone/linein     */

 ulError = mciSendCommand( mciOpenParms.usDeviceID,
                           MCI_CONNECTOR,
                           MCI_WAIT | MCI_CONNECTOR_TYPE |
                           MCI_ENABLE_CONNECTOR,
                           (PVOID) &mciConnectorParms,
                           0 );
 if (ulError)
  {
    ShowMCIErrorMessage( ulError);
    return( FALSE);
  }
```

Sequencer Device

The OS/2 sequencer device plays a MIDI song by sending commands from a MIDI file to a synthesizer, where the commands are converted to the sounds of a specific instrument. Typically, a digital signal processor (DSP) is used to generate the sounds of the instrument, which results in an authentic reproduction of the original performance.

General MIDI (Musical Instrument Digital Interface) is a standard specification for playing back music from a series of commands, rather than actual audio data. The commands represent musical events, such as turning a note on and off ("Note On" and "Note Off"), as well as timing mechanisms for specifying the duration of the note sound. The sequencer uses the timing commands to sequence the playing of the music.

Following is a text example of the commands generated when someone depresses the "middle C" key of a synthesizer keyboard: a note-on command X'90' and two bytes of data:

```
X'90' - "Note-on" command to MIDI channel 0
X'3C' - Keyboard note (middle-C)
X'40' - Velocity (X'00'-X'7F').
```

MIDI augments waveform audio as a means of producing sounds in the multimedia environment. MIDI data offers the advantage of requiring far less storage than waveform data. For example, suppose a three-note chord-middle-C, E and G- is held for one second. Following are the MIDI commands required to reproduce the chord on a synthesizer:

```
TIME = 0 sec.  90 3C 40  90 40 40  90 43 40
               (C-on)    (E-on)    (G-on)
.
.
.
TIME = 1 sec.  90 3C 00  90 40 00  90 43 00
               (C-off)   (E-off)   (G-off)
```

The storage required for the MIDI commands is 18 bytes. To store the same information as 16-bit, PCM, 44 kHz, stereo waveform audio data requires 176KB.

Another advantage of storing musical performances as a series of instructions is that the information can be edited, the same way words in a document can be edited by a word processor. The musical editing process can be used, for example, to correct mistakes in an artist's original interpretation, or to change certain points of style before playback or final recording. Playback of MIDI data using the sequencer media device can be used to reproduce the original performance or to print out musical scores.

Sequencer Device Specifics

The sequencer device sends MIDI messages and data to the audio adapter. Some audio adapters, such as the M-Audio adapter, perform FM synthesis to produce music. Other audio adapters, such as the Sound Blaster adapter, have the capability to send the MIDI data through a MIDI port to an external synthesizer device.

The OS/2 sequencer device does not currently support recording new MIDI information.

MIDI Stream Connector

The *MIDI stream* connector represents the flow of MIDI information from the sequencer device to its associated amplifier-mixer (ampmix) device. During playback, the sequencer device sends MIDI information from either application memory or files to the ampmix device for subsequent conversion into audio that can be heard through conventional speakers or headphones.

Control of the characteristics of the MIDI information is provided by the sequencer device. Volume control is provided as an additional service, although this feature is actually provided by the ampmix device in a way that is transparent to the calling application. If other advanced audio-shaping features are required, the application can retrieve the device ID of the ampmix device using the MCI_CONNECTION message. Once the device ID has been obtained, the application can send **set** commands directly to the ampmix device to manipulate audio attributes, such as treble, bass, balance, and so on.

MIDI Data Formats

The MIDI support for the OS/2 multimedia system handles the RIFF RMID data type, as well as standard MIDI file formats 0 and 1. MIDI file format 0 merges tracks of MIDI data into one track; MIDI file format 1 preserves the separate tracks of data. MIDI file format 2 is not supported.

MIDI files are made up of chunks, similar to RIFF chunks. In MIDI files, there are two types of chunks: header chunks and track chunks. A *header* chunk provides a minimal amount of information pertaining to the entire file. A *track* chunk contains a sequential stream of MIDI data, which can contain information for up to 16 MIDI channels.

General MIDI Specification

The standard MIDI channel, patch, and percussion key assignments shown in the following tables are defined in the General MIDI Specification issued by the MIDI Manufacturers Association (MMA). A detailed specification can be ordered from:
International MIDI Association
5316 West 57 Street
Los Angeles, CA 90056

Channel Assignments

Channels are divided into two general categories, low-end synthesizer support and high-end synthesizer support. The following table shows the channel assignments for this standard.

Channel Range	Use Description	Polyphony
1 through 9	Extended Melodic Tracks	16 Notes
10 Only	Extended Percussion Track	16 Notes
11 through 12	Unused Tracks	
13 through 15	Base-Level Melodic Tracks	6 Notes
16 Only	Base-Level Percussion Track	3 Notes

Patch Assignments

The following table shows the standard patch definitions for MIDI instruments. Each family of instruments (for example, strings or brass) has eight different voice numbers reserved for patch definitions.

Piano	Chromatic Percussion	Organ	Guitar
0 Acoustic Grand Piano	8 Celesta	16 Hammond Organ	24 Acoustic Guitar (nylon)
1 Bright Acoustic Piano	9 Glockenspiel	17 Percussive Organ	25 Acoustic Guitar (steel)
2 Electric Grand Piano	10 Music box	18 Rock Organ	26 Electric Guitar (jazz)
3 Honky-Tonk Piano	11 Vibraphone	19 Church Organ	27 Electric Guitar (clean)
4 Rhodes Piano	12 Marimba	20 Reed Organ	28 Electric Guitar (muted)
5 Chorused Piano	13 Xylophone	21 Accordion	29 Overdriven Guitar
6 Harpsichord	14 Tubular Bells	22 Harmonica	30 Distortion Guitar
7 Clavinet	15 Dulcimer	23 Tango Accordion	31 Guitar Harmonics
Bass	Strings	Ensemble	Brass
32 Acoustic Bass	40 Violin	48 String Ensemble 1	56 Trumpet
33 Electric Bass (finger)	41 Viola	49 String Ensemble 2	57 Trombone
34 Electric Bass (pick)	42 Cello	50 Synth Strings 1	58 Tuba
35 Fretless Bass	43 Contrabass	51 Synth Strings 2	59 Muted Trumpet
36 Slap Bass 1	44 Tremolo Strings	52 Choir Aahs	60 French Horn
37 Slap Bass 2	45 Pizzicato Strings	53 Voice Oohs	61 Brass Section
38 Synth Bass 1	46 Orchestral Harp	54 Synth Voice	62 Synth Brass 1
39 Synth Bass 2	47 Timpani	55 Orchestra Hit	63 Synth Brass 2
Reed	Pipe	Synth Lead	Synth Pad
64 Soprano Sax	72 Piccolo	80 Lead 1 (square)	88 Pad 1 (New Age)
65 Alto Sax	73 Flute	81 Lead 2 (sawtooth)	89 Pad (warm)

66 Tenor Sax	74 Recorder	82 Lead 3 (calliope lead)	90 Pad 3 (polysynth)
67 Baritone Sax	75 Pan Flute	83 Lead 4 (chiff lead)	91 Pad 4 (choir)
68 Oboe	76 Bottle Blow	84 Lead 5 (charang)	92 Pad 5 (bowed)
69 English Horn	77 Shakuhachi	85 Lead 6 (voice)	93 Pad 6 (metallic)
70 Bassoon	78 Whistle	86 Lead 7 (fifths)	94 Pad 7 (halo)
71 Clarinet	79 Ocarina	87 Lead 8 (bass + lead)	95 Pad 8 (sweep)
Synth Effects	Ethnic	Percussive	Sound Effects
96 FX 1 (rain)	104 Sitar	112 Tinkle Bell	120 Guitar Fret Noise
97 FX 2 (soundtrack)	105 Banjo	113 Agogo	121 Breath Noise
98 FX 3 (crystal)	106 Shamisen	114 Steel Drums	122 Seashore
99 FX 4 (atmosphere)	107 Koto	115 Woodblock	123 Bird Tweet
100 FX 5 (brightness)	108 Kalimba	116 Taiko Drum	124 Telephone Ring
101 FX 6 (goblins)	109 Bagpipe	117 Melodic Drum	125 Helicopter
102 FX 7 (echoes)	110 Fiddle	118 Synth Drum	126 Applause
103 FX 8 (sci-fi)	111 Shanai	119 Reverse Cymbal	127 Gunshot

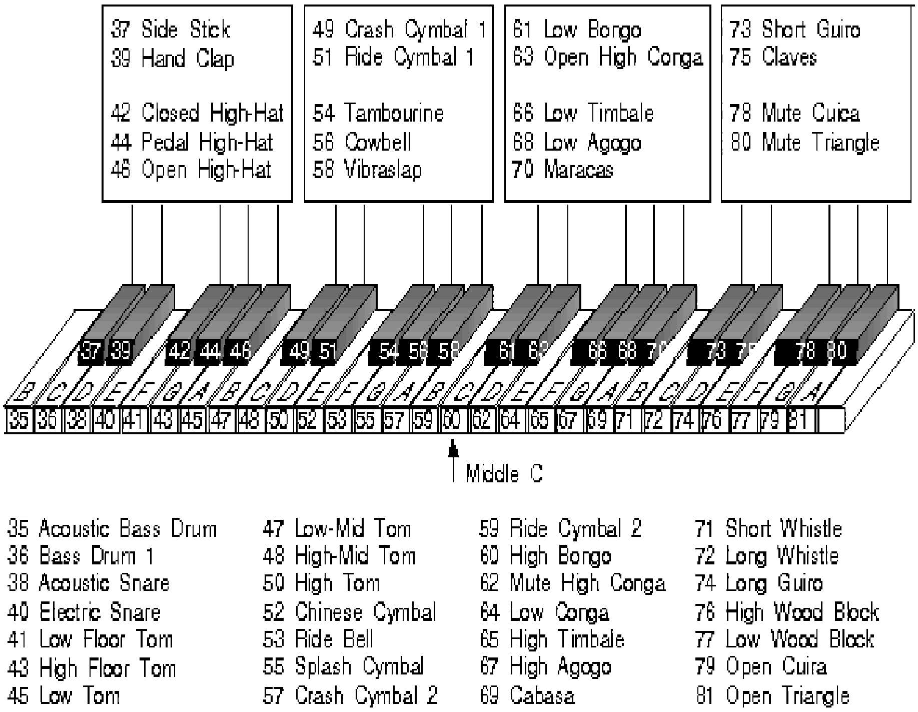
Percussion Key Assignments

In MIDI, timbre (the sound of a specific instrument; for example, a violin or a trumpet) for most instruments is assigned a specific number. For example, voice 56 represents a trumpet sound. Thus, all note values for voice 56 produce notes having the distinctive sound of a trumpet.

Typically, number assignments for percussion instruments are handled differently. Although each percussion instrument has a distinctive sound, all percussion instruments for a particular synthesizer are assigned one voice or timbre number. For example, if a synthesizer specifies that timbre 45 is a PCM percussion set, then note 60 (Middle C) might be for a kettle drum, 61 for a bass drum, 62 for a triangle and so on.

Many manufacturers use percussion and note number assignments that are unique to their hardware. For example, note 60 might be assigned to a kettle drum on one synthesizer model and to castanets on another.

The following figure shows the standard percussion key definitions.

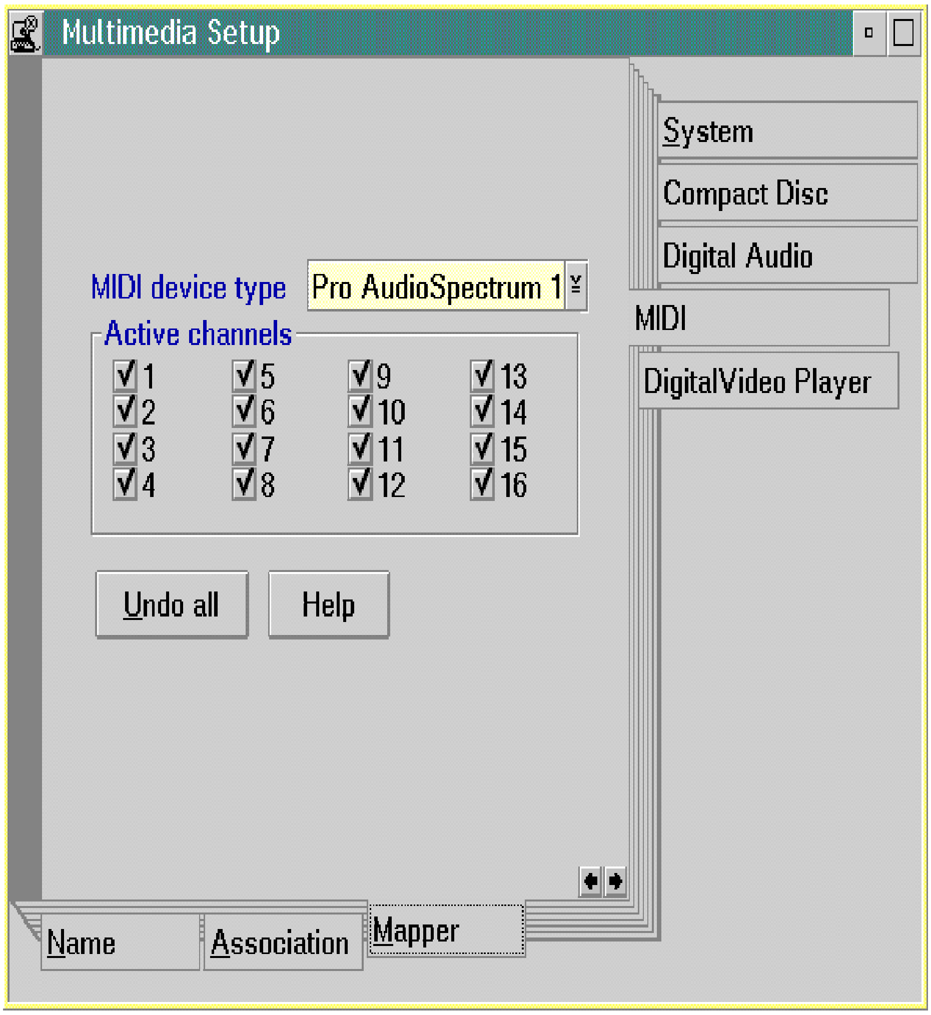


MIDI Mapping Function

The MIDI mapping function provided with the sequencer device offers a level of device independence to application developers. Because MIDI patch assignments and percussion key assignments of different MIDI manufacturers vary, the MIDI mapper can be enabled to dynamically translate MIDI data in real-time, as a MIDI song is played. An application enables the mapper by setting the port with the MCI_SET command:

```
set sequencer port mapper
```

The user can configure the sequencer device by selecting from a list of available device types. The list is provided on the mapper page of the Multimedia Setup application. OS/2 multimedia provides a General MIDI map, as well as a map for each sequencer device supported by OS/2 multimedia. A mapper page (shown below) also allows the user to enable or disable specific channels.



MIDI device type: Pro AudioSpectrum 1

Active channels

☑ 1 ☑ 5 ☑ 9 ☑ 13
☑ 2 ☑ 6 ☑ 10 ☑ 14
☑ 3 ☑ 7 ☑ 11 ☑ 15
☑ 4 ☑ 8 ☑ 12 ☑ 16

Once the MIDI mapper is enabled, the Sequencer expects the MIDI files it plays to conform to the General MIDI Specification. The MIDI mapper translates the General MIDI format to the appropriate device format as specified in the Mapper page of a MIDI device in Multimedia Setup.

Guidelines for MIDI Song Authors

MIDI authors should be aware of several design concerns when producing a MIDI song:

- The General MIDI specification defines a base-level configuration and an extended-level configuration for synthesizers. Because you cannot control which synthesizer is used to play your song, you should provide percussion and melody tracks for both configurations. The table in the Channel Assignments section shows you which channels to allocate.

- Put your important melody sounds in the lower-numbered channels. By prioritizing channel use, you ensure that your song sounds reasonable when it is played on hardware that supports only a few channels.

- When selecting instruments for your MIDI song, use the instruments that are defined for the most popular MIDI devices. Your song can then be easily and accurately mapped to other hardware formats.

- When using nonpercussive channels, limit the polyphony (number of simultaneous notes) to 6 notes for the base level and 16 notes for the extended level.

- When using percussion channels, limit the polyphony to 3 notes for the base level and 16 notes for the extended level.

- Use the standard General MIDI patch assignments and percussion key assignments shown in the table in the Patch Assignments section and the figure in the Percussion Key Assignments section.

- Always send a program change command to a channel before sending other commands to the channel. For channels 10 and 16, which are used for percussion, select patch 0.

- Always send a MIDI main volume controller command (controller number 7) to a channel after selecting a patch by sending a program change command. Use the value of 80 (X'50') for normal listening levels.

Using the Sequencer Device

After you open the sequencer device, query the *division type* of the device element with a **status** command. Division type refers to the method used to represent the time between MIDI events in the sequence.

```
open mysong.mid alias midi1 shareable
status midi1 division type wait
```

A MIDI file's division type can be either PPQN or any of the following SMPTE formats:

PPQN (parts-per-quarter-note)
SMPTE 24 frame
SMPTE 25 frame
SMPTE 30 frame
SMPTE 30 drop frame

After you determine the file division type, you can make other **status** queries such as:

Query	**Response**
length	Length of sequence
length track *n*	Length of track *n*
position	Current position of sequence
position track *n*	Current position of track *n*
tempo	Current tempo.

Responses are all in the current time format. PPQN files return length and position information in song pointer units. However, SMPTE files return the information in colon format HOURS:MINUTES:SECONDS:FRAMES.

PPQN files return the tempo in beats per minute; SMPTE files return the tempo in frames per second.

Playing A MIDI Song

Before you start playing MIDI music, you may want to set the port to the MIDI mapper so that channel and patch reassignments can be made.

```
set midi1 port mapper
seek midi1 to start wait
play midi1 notify
.
.
.
** playing **
.
.
.
close midi1
```

Creating MIDI Memory Files

Applications that access memory buffers to store and access MIDI data can use the memory I/O features of the multimedia input/output (MMIO) file services. This technique consists of opening a memory file using mmioOpen. mmioOpen has a pointer to the buffer of MIDI data as a parameter. This buffer can then be operated on by MCI and MMIO as if it were a file.

The details of opening the file and setting up the MIDI buffer varies depending on an application's requirements. For example, the memory buffer can be allocated by the MMIO system and filled subsequently by the application, or the memory buffer can be allocated by the application and passed to MMIO. The buffer can be filled in different ways such as mmioRead, mmioWrite, and mmioAdvance.

Care must be taken when calling MMIO functions and sending MCI messages to the same memory file. MMIO and MCI are independent subsystems linked only through the MMIO memory handle passed to MCI_OPEN. There is for example, no relationship between MCI_SEEK and mmioSeek. Each subsystem keeps its own set of relevant files and stream pointers. If one subsystem changes the data in memory, but the memory had previously been cued with MCI_CUE, the change of data will not be recognized by MCI until a call to reload the streams has been issued.

The following code fragment shows the opening of a memory file with a user-supplied MIDI buffer of untranslated (format 0 or 1) data and the playing of that data through MCI.

```
{
   /* variable for IOProc */
   PMMIOPROC  pIOProc;
   HMODULE    hModMidiio;

   /* variables for memory file */
   MMIOINFO  mmioInfo;
   CHAR      UserBuffer[SIZE_OF_BUFFER];
   HMMIO     hmmio;

   /* variables for MCI commands */
   MCI_OPEN_PARMS  mop;
   MCI_PLAY_PARMS  mpp;


   /* Open memory file. Provide MIDI-filled data buffer to MMIO, so
    * data buffer becomes file image in memory.  Also specify that
    * the data will need to be translated.
    */

 mmioInfo.pchBuffer = UserBuffer; /* Filled with untranslated
                                     MIDI data                */
 mmioInfo.cchBuffer = SIZE_OF_BUFFER; /* User-defined         */
 mmioInfo.ulTranslate = MMIO_TRANSLATEDATA | MMIO_TRANSLATEHEADER;
                                 /* Need to translate data    */
 mmioInfo.fccIOProc = mmioFOURCC( 'M', 'I', 'D', 'I');/* Data
                                                          format */
 mmioMemInfo.fccChildIOProc = FOURCC_MEM;        /* Storage type */
 hmmio = mmioOpen ( NULL, mmioInfo, MMIO_READWRITE );

    /* open MIDI device */

    mop.pszElementName = (PSZ) hmmiomem;

    mciSendCommand(
         0,                        /* We don't know the device yet. */
         MCI_OPEN,                 /* MCI message                   */
         MCI_WAIT | MCI_OPEN_MMIO |
         MCI_OPEN_TYPE_ID | MCI_OPEN_SHAREABLE
         (ULONG) &mop,         /* Parameters for the message    */
         0 );                      /* Parameter for notify message  */

    /* play MIDI memory file for 1 second */

    mpp.ulFrom=0;
    mpp.ulTo=3000;    /* default is MMTIME units (1/3000 second) */
    mciSendCommand(
            mop.usDeviceID,        /* Device to play the data    */
            MCI_PLAY,              /* MCI message                */
            MCI_WAIT |
            MCI_FROM | MCI_TO,     /* Flags for the MCI message  */
            (ULONG) &mpp,          /* Parameters for the message */
            0 );                   /* No parm necessary          */

     /* close device */
     mciSendCommand(
            mop.usDeviceID,        /* Device to play this          */
            MCI_CLOSE,             /* MCI message                  */
            MCI_WAIT,              /* Flags for the MCI message    */
            (ULONG) NULL,          /* Parameters for the message   */
            (ULONG) NULL );        /* Parameter for notify message */

}
```

Sequencer Command Messages

Message	Description
MCI_CLOSE	Closes the sequencer device.
MCI_CONNECTOR	Enables or disables a connector, queries its state or identifies its type.
MCI_CUE	Cues the device for minimum delay in playback or recording: -Cues a sequencer input device -Cues a sequencer output device
MCI_GETDEVCAPS	Gets device capabilities.
MCI_INFO	Gets the following information: -Sequencer product name -Name of the currently loaded file
MCI_LOAD	Loads a sequencer data file.
MCI_OPEN	Initializes the sequencer device.
MCI_PLAY	Plays back MIDI data by means of the audio adapter. The following optional action modifies MCI_PLAY: -Specify start and stop positions in the device element
MCI_PAUSE	Suspends the current playback action.
MCI_RESUME	Resumes playing from a paused state, keeping previously specified parameters in effect.
MCI_SAVE	Saves the device element in its current format.
MCI_SEEK	Moves to the specified position in the device element.
MCI_SET	Sets audio attributes: -Identify the channels to be used -Set the volume -Enable or disable the audio output Sets sequencer information in the PMCI_SEQ_SET_PARMS structure of MCI_SET: -Set the sequencer as master and specify the type of MIDI data it uses -Change the SMPTE offset of a sequence -Set the output MIDI port of the sequencer -Set the MIDI mapper as the port to receive sequencer messages -Set the sequencer as slave and identify any sync data it needs

	-Set the tempo of the MIDI sequence according to the current time format Sets time format to be used for sequencer data: -Song pointer -SMPTE 24 -SMPTE 25 -SMPTE 30 -SMPTE 30 drop
MCI_SET_CUEPOINT	Sets a cue point.
MCI_SET_POSITION_ADVISE	Sets a position change notification request.
MCI_SET_SYNC_OFFSET	Sets a synchronization offset.
MCI_STATUS	Receives status on items such as the following: -File division type -Length of sequence in current time format -Whether the device is the sync master or a slave -Which MIDI port is assigned to the sequencer -What the offset is for a SMPTE-based file -What the current tempo and time format is
MCI_STOP	Stops the sequencer device before loading a file.

Sequencer Connectors

The sequencer device directly supports one *MIDI stream* connector that is always enabled. As it is likely that an application will need to select the output destination on the amplifier-mixer device, the sequencer device will attempt to provide the following connector services to an application. If the requested connector is not available, the command will fail.

- headphones
- speakers
- line out

Additional connectors may be available on the ampmix device. To control these connectors, obtain the device ID of the ampmix device using the MCI_CONNECTION message and issue the connector command directly to the associated amplifier-mixer.

To determine which connectors are supported by an amplifier-mixer device, use the MCI_CONNECTORINFO message.

CD Audio Device

The CD audio media device provides access to devices that read compact discs for the purpose of playing CD audio. A typical use for CD audio is to provide high quality audio for use in a presentation. Another use of CD audio would be to provide detailed audio help for an application user. Instead of the usual hyperlinked text and graphics, an entire step by step audio tutorial might be stored on a compact disc in several different languages.

Compact Disc Formats

A compact disc can contain several different kinds of information. Many compact discs appear to be very similar to other forms of permanent storage such as hard or floppy disks. CD-ROM (Compact Disc - Read Only Memory) refers to a file format used for compact discs containing data that can be read by a computer's file system. This format is also known as Yellow Book. *Yellow Book* refers to the specification published by Philips Consumer Electronics Company and Sony Corporation that describes the physical layout of the CD-ROM disc format. This document evolved into the ISO (International Organization for Standardization) 9660 standard.

Another form of compact disc data is Compact Disc Digital Audio (CD-DA) and is also known as Red Book. This format is used for compact discs containing digital audio. CD-DA discs are encoded as 16-bit stereo PCM (Pulse Code Modulation) at 44.1 kHz. See Waveform Data Formats for more information on this recording method.

For every second of CD-DA audio data, 172KB of disc storage is required. Therefore, the sustained data rate required to play back a selection from a CD-DA track is 172KB per second. The CD-DA track format typically has an error recovery rate of 90 percent. A 10 percent error rate is acceptable for audio discs, because errors that escape the correction mechanism are usually compensated for by audio filters in the CD player hardware.

Unlike CD-DA data, information stored in a CD-ROM file system cannot tolerate such a relatively high error rate. As a result, CD-ROM format data is transferred to an application at an apparent data rate of 150KB per second. The lower data rate results from the additional error correction bits utilized by the file system to produce an acceptable error rate.

Mixed Format Compact Discs

A mixed format compact disc holds CD-ROM file system data as well as CD-DA track format data. An example of the use of a mixed format disc is an application that contains several symphonies by a famous composer. The actual audio is stored as a series of CD-DA tracks. Also stored on the disc (but in CD-ROM file format) is a program, the actual music score, and perhaps a data base on the composer's life. When the application is started, the audio from a symphony can be played using the CD audio media driver, while the user is allowed to study the music score. Additionally, the user might retrieve facts on the composer, such as how old the composer was when the symphony was written.

By issuing the MCI_GETTOC message with mciSendCommand, an application can retrieve a table of contents (TOC) for a disc. This function is particularly useful for a mixed format disc. The starting and ending addresses (time) of each audio track are listed as well as the type of track. Using the table of contents, an application can immediately determine which tracks can be played using the CD audio device. Additionally, an application can determine the track type by issuing the **status** command to determine the type of any individual track.

```
open cdaudio alias mycd shareable
status mycd type track 1 wait
```

Note: The CD audio device processes only CD-DA tracks.

CD-ROM Drives and Streaming

Depending on the type of CD-ROM drive installed, the audio data on a CD-DA disc is either processed by a Digital-to-Analog Converter (DAC) that is built into the drive, or it is moved through the system to a Digital Signal Processor (DSP) on an audio adapter. Some CD-ROM drives can only play CD-DA audio data through the built-in DAC. Others, like the IBM Personal System/2 (PS/2) CD-ROM-II Drive, can play through the DAC, or they can stream data through the audio adapter DSP.

The advantage offered by playing CD-DA through the DAC is that it is a simple operation that greatly reduces system and resource overhead. The advantage gained by streaming data through an audio adapter DSP is that you can potentially enhance the signal beyond the capabilities of the DAC by adding special effects such as reverberation and tremolo, or by modifying treble and bass-capabilities the DAC cannot provide.

The default mode for the CD audio device is playback processed by the DAC through the internal headphones connector. To switch to playback through the audio adapter DSP, the application sends the **connector** command enabling the *cd stream* connector.

```
connector mycd enable type cd stream wait
```

To resume playback through the internal CD DAC, the application should re-enable the *headphones* connector. This will automatically disable the streaming connector.

```
connector mycd enable type headphones wait
```

Contention for the Amplifier-Mixer Device

Because the audio adapter is modeled as an amplifier-mixer (ampmix) device, the CD audio device will automatically open and connect to its default amplifier-mixer when streaming CD-DA data. If the amplifier-mixer device is currently opened exclusively by another application, enabling the cd stream connector will fail. If the amplifier-mixer is available, then other applications using the same ampmix device might temporarily lose use of the device if the audio adapter cannot simultaneously process all requests. The Media Device Manager (MDM) will handle all resource allocation automatically, however it should be realized that streaming CD-DA data using the CD audio device can effect applications using the waveaudio and sequencer devices as these devices also use the ampmix device. Conversely, a waveaudio device could cause the loss of a streaming CD audio device if the waveaudio device requires the use of the same shared ampmix device.

Using the CD Audio Device

The CD audio device is a dynamic single context device and is serially shareable.

```
open cdaudio alias mycd shareable wait
```

By setting the **shareable** flag, an application allows other applications to share the device. When the device context is about to become active, the multimedia system posts an MM_MCIPASSDEVICE message with an event of MCI_GAINING_USE to your application.

Another common aspect of using the CD audio device is controlling the volume. Volume is controlled by indicating a percentage of the maximum achievable effect. Like all other devices in OS/2 multimedia, this volume level is automatically tempered by the master volume level that was set using the Volume Control application.

Volume Control Using the Internal DAC

By default the CD audio device utilizes the internal DAC on the CD drive which is represented by an enabled headphones connector. When the internal DAC is being used to process the CD audio data, requests to set the volume should be sent directly to the CD device.

```
set mycd audio volume 50 all wait
```

Depending on the capabilities of the actual CD-ROM drive, the degree of volume control might vary from on or off, to a reasonably linear range of settings. For example, the IBM PS/2 CD-ROM-II Drive supports 16 different levels of volume. An application might want to display either a simple two state mute button or a volume slider or dial depending on the degree of volume control provided by the CD drive. If a CD audio device can set the volume to some value other than 0 or 100, then it is likely the device supports several volume levels. For more information on two-state graphical buttons, see OS/2 Multimedia Controls. Refer to the *PM Programming Reference* for information on linear sliders and circular sliders (dials).

Volume Control Using the Ampmix Device

As stated previously, some CD audio devices have the capability to stream the audio data to a connected amplifier-mixer device. To determine if a particular CD audio device can stream CD audio data the device capabilities command can be used. If TRUE is returned, then the *cd stream* connector can subsequently be enabled.

```
capability mycd can stream wait
.
.
.
** If this CD audio device can stream **
.
.
.
connector mycd enable type cd stream wait
```

After enabling the cd stream, the CD audio device is utilizing the amplifier-mixer device to convert the audio data into sound. As such, the application must now send all volume commands to the connected ampmix device. Before a command can be sent to the ampmix device, the application must obtain the device ID of the amplifier-mixer device using the MCI_CONNECTION message. If using the string interface, an alias can be assigned to the connected ampmix device as follows:

```
connection mycd query type cd stream alias amp wait
```

In the previous example, the alias "amp" is assigned to the connected ampmix device. To set the volume or to control other audio attributes, the application can now send messages directly to the ampmix device.

```
set amp audio volume 25 over 2000 all wait
set amp audio treble 50 wait
```

Playing a Compact Disc

Before issuing a play command, suitable media should be present in the CD drive. The following **status** command will return TRUE if the disc contains CD-DA tracks that can be played by the CD audio device:

```
status mycd media present wait
```

The CD audio device operates only on discs that contain CD-DA tracks. If a disc contains no CD-DA tracks, then the MCIERR_INVALID_MEDIA error can be returned on any command that requires a CD-DA track format in order to complete. Example commands include **play**, **seek**, and the *media present* function of the **status** command.

Other pertinent information regarding the CD audio device can be obtained using the **status** command:

Query	**Response**
ready	TRUE or FALSE
mode	not ready, open, paused, playing, seeking, or stopped
time format	milliseconds, MMTIME, MSF, TMSF
volume	Current volume setting

The **status** command will also return the following information about the currently inserted media:

Query	**Response**
current track	Number of current track
position in track	Current position relative to track start
length track *n*	Length of track *n*
position track *n*	Starting position of track *n*
type track *n*	Audio or data
copypermitted track *n*	TRUE, if digital copying permitted
channels track *n*	Number of audio channels on track
preemphasis	TRUE, if track was recorded with preemphasis
start position	Starting position of the disc
position	Position in current time format
number of tracks	Number of audio tracks on the disc
length	Total length of tracks on the disc

Changing the Media

It is the responsibility of the application to ensure that the appropriate compact disc is in the CD drive if having a particular disc is essential to the application. For example, a CD player application might simply update its track and time displays if a new disc is inserted. Other applications might be so dependent on a specific disc that the user must be prompted to re-insert the appropriate disc. An application can choose to disable the manual eject button on the physical CD drive to prevent the disc from being changed.

```
capability mycd can lockeject wait
.
.
.
** If the door can be locked, lock it! **
```

```
.
.
.
set mycd door locked wait
```

If the drive does not support disabling the manual eject, then the application can check the disc identity by obtaining the UPC code (serial number) or the more general *CD ID* . The CD ID is an 8-byte identifier which can be obtained using the **info** command and is constructed from the following information:

- Starting track address
- Ending track number
- Lead-out track address.

```
info mycd ID wait
```

The UPC code is a serial number which has been assigned to a particular compact disc and can also be obtained using the **info** command, however not all disc manufacturers utilize a UPC code. The UPC code is represented as a binary coded decimal (BCD) number.

```
info mycd UPC wait
```

If a play is in progress and the manual eject button is pressed on the CD drive, the application will receive the MCIERR_DEVICE_NOT_READY error when the play command is aborted.

Unique Considerations for Streaming

When the OS/2 multimedia system is streaming multimedia information from a CD-ROM drive, attempts by other applications to access and control the CD drive for normal file system operations can be suspended until the streaming operation is ended. The stream can be interrupted by a pause or a stop command, the completion of playback, or the ejection of the disc. Once the stream is interrupted, any applications that were waiting can resume their attempts to gain control of the device.

If the CD audio device is being shared and the disc is changed on an application which had lost use of the device, any active play command may be aborted with an error of MCIERR_MEDIA_CHANGED when use of the device is subsequently regained.

CD-DA Command Messages

Message	Description
MCI_OPEN	Initializes the CD-DA device.
MCI_GETDEVCAPS	Gets device capabilities.
MCI_GETTOC	Gets a table of contents structure for the currently loaded disc.
MCI_CUE	Cues the device for minimum delay in playback.
MCI_PLAY	Starts playing audio data from the disc. The following optional action modifies MCI_PLAY: -Specify start and stop positions on the disc
MCI_PAUSE	Suspends the current playback action.
MCI_RESUME	Resumes playing from a paused state, keeping

	previously specified parameters in effect.
MCI_SEEK	Moves to the specified position on the disc.
MCI_SET	Sets audio attributes: -Identify the channels to be used -Set the volume -Apply the audio attribute change over a period of time (fade) -Enable or disable the audio output Retract the tray and close the door, if possible -Open the door and eject the tray, if possible -Set the time format in milliseconds, MSF, TMSF, or MMTIME
MCI_SET_SYNC_OFFSET	Specifies positional offsets.
MCI_STATUS	Receives status on items such as the following: -Current volume setting -Length of the disc -Whether the media is present in the device -Current mode of the device; for example, "stopped" -Current position in the media
MCI_INFO	Fills a user-supplied buffer with the following information: -Product name and model number of the current audio device -Serial number (UPC) of the current disc -ID of the current disc
MCI_STOP	Stops playing the CD-DA device.
MCI_SET_CUEPOINT	Sets a cue point.
MCI_SET_POSITION_ADVISE	Sets a position change notification request.
MCI_CLOSE	Closes the CD-DA device.
MCI_CONNECTOR	Enables or disables a connector, queries its state or identifies its type.

CD Audio Connectors

The number and type of connectors can vary by manufacturer. To determine which connectors are supported, an application can issue the MCI_CONNECTORINFO message. Also, the device capabilities command can be issued to determine if the CD drive can stream audio to an amplifier-mixer device. The following are typical of the connectors supported by CD audio devices:

- Headphones
- CD stream

CD-XA Device

CD-XA Disc Formats

The CD-XA media driver provides access to devices that support CD-ROM/XA discs. CD-XA (Compact Disc-Extended Architecture) refers to a storage format that accommodates interleaved storage of audio, video and standard file system data. CD-XA data is stored in a file system format on the discs, and playback control is managed by the CD-XA media device in cooperation with the amplifier-mixer device.

CD-XA takes advantage of a special ADPCM audio compression mechanism that not only yields a low data rate but also enables more audio data to be stored on a disc than that allowed by a CD-DA disc. ADPCM (Adaptive Delta Pulse Code Modulation) is an audio compression technique that allows up to a 16 to 1 compression of audio data.

By compressing the audio data (in some cases to 1/16 the size of CD-DA data) it now becomes possible to record multiple audio tracks on a single disc. With CD-XA level C, recorded in stereo, it is possible to interleave 8 different audio tracks on a single disc. With CD-XA level C, recorded in mono, this number climbs to 16 different tracks on a single disc.

The following table summarizes the different compression levels available with CD-XA and compares them to CD-DA.

Type	Audio	Video	File Sys	Data Rate per Sec	One Sec of Audio Data	Spec
CD-DA	16-bit PCM at 44.1 kHz	No	No	172KB	Stereo: 172KB	Red Book
CD-DA+G	16-bit PCM at 44.1 kHz	288 x 192 CLUT4	No	172KB	Stereo: 172KB	Red Book
CD-ROM	No	No	Yes	150KB	N/A	High Sierra, ISO 9660, Yellow Book
CD-XA Level B	4-bit ADPCM at 37.8 kHz	320 x 200 640 x 480 CLUT1, 4 or 8	Yes	150KB	Mono: 18.75KB Stereo: 37.5KB	Green Book
CD-XA Level C	4-bit ADPCM at 18.9 kHz	320 x 200 640 x 480 CLUT1, 4 or 8	Yes	150KB	Mono: 9.4KB Stereo: 18.75KB	Green Book

----

CD-XA Data Types

The CD-XA media device can play four streams of CD-XA data types simultaneously from the same CD-XA file. CD-XA data streams are set up by specifying the following flags with MCI_SET:

Flag	**CD-XA Stream Data Type**
MCI_CDXA_AUDIO_DEVICE	CD-XA audio channel to an audio adapter.
MCI_CDXA_AUDIO_BUFFER	CD-XA audio channel to an audio memory playlist.
MCI_CDXA_VIDEO_BUFFER	CD-XA video channel to a video memory playlist.
MCI_CDXA_DATA_BUFFER	CD-XA data channel to a data memory playlist.

In the future IBM may provide the capability of setting up additional streams to more than one audio device, or more than one audio, video, or data buffer playlist.

If a channel contains data of more than one CD-XA data type (data and video), and an MCI_SET is done, only data of the type specified is returned. For example, suppose an MCI_SET message is sent with MCI_CDXA_VIDEO_BUFFER, channel 0, and a playlist specified. Although channel 0 contains sectors of both audio and video, only the video sectors are sent to the memory playlist buffers specified with the MCI_SET. If your application wants to retrieve the audio sectors from channel 0, it must do another MCI_SET, this time specifying MCI_CDXA_AUDIO_BUFFER, channel 0, and a playlist that is different from the one specified for the video data. The audio would then be sent to the playlist specified for MCI_CDXA_AUDIO_BUFFER, and the video sectors would be sent to the playlist specified for MCI_CDXA_VIDEO_BUFFER.

Using the CD-XA Device

When an MCI_OPEN or MCI_LOAD request is made, it is recommended that a drive letter not be included as part of the element name. If a drive letter is specified, it must match the drive letter of the open device; otherwise, the error MCI_FILE_NOT_FOUND is returned.

Setting Up the Primary Stream

The first stream is set up by calling MCI_SET with one of the four target specifications:

MCI_CDXA_AUDIO_DEVICE
MCI_CDXA_AUDIO_BUFFER
MCI_CDXA_VIDEO_BUFFER
MCI_CDXA_DATA_BUFFER

The first stream to be set up is called the *primary* stream, because it controls much of the operation of the CD-XA device. When stream control operations are done on the CD-XA media device, they are done to the primary stream. For example, when an MCI_SEEK is done, only the primary stream is seeked. This makes sense because the media is the same for all streams, and seeking one of the streams affects the disc position for all the other streams.

Because the primary stream is the control focus for the entire device, if an MCI_SET is done while the MCI_SET_OFF flag is in effect for the primary stream, the device is stopped. If the MCI_SET is done on a secondary stream, only that secondary stream is stopped.

If an MCI_SET with MCI_SET_ON is not done before an MCI_PLAY, MCI_CUE, or MCI_SEEK is sent, the first audio channel found on the CD-XA file is used as the default and is set up as the primary stream using the MCI_CDXA_AUDIO_DEVICE. It is recommended that an MCI_SET be sent for the primary stream before an MCI_PLAY, MCI_CUE or MCI_SEEK. It is also recommended that MCI_CDXA_AUDIO_DEVICE be used as the primary stream, because seeking and playing with FROM and TO positions specified can only be done on a stream to an audio device. The video, data and audio buffers do not have seek or time capability. When CD-XA data types are streamed to memory playlists, this operation is done as a data transport and does not have a real timing aspect associated with it. For example, if an MCI_SET with an MCI_CDXA_DATA_BUFFER stream was set as the primary stream, a seek or a status position to an MMTIME unit or millisecond does not make sense, because there is no correlation of time to the data volumes.

When an MCI_SET with MCI_CDXA_AUDIO_DEVICE is done, the audio quality level is set by the media device, based on the quality level found in the first sector. The audio quality level must be constant within a channel within a single file when streamed to an audio device. The current CD-XA software does not detect any deviations that occur. If the quality level changes in a channel, the output audio may be garbled and the stream time will be inaccurate.

If the channel associated with the primary stream is set off and a PLAY command is received, the error MCIERR_CHANNEL_OFF will be returned.

The valid range for audio channels is 0-15. The valid range for video and data channels is 0-31. These ranges are defined by CD-XA specifications. MCIERR_OUTOFRANGE is returned if the channel specified on the MCI_SET is outside the range.

The channel assigned to a secondary target specification (MCI_CDXA_AUDIO_DEVICE, MCI_CDXA_AUDIO_BUFFER, MCI_CDXA_VIDEO_BUFFER, or MCI_CDXA_DATA_BUFFER) can be changed without closing the CD-XA media device. To do this, your application calls MCI_SET with the MCI_SET_OFF flag for that channel and then calls MCI_SET with the MCI_SET_ON flag and the new channel number. If MCI_SET with the MCI_SET_ON flag is called for a target specification already in use, MCIERR_RESOURCE_NOT_AVAILABLE is returned if the system or underlying hardware cannot support the additional resource demands.

The channel assigned to the primary target specification can be changed just like a secondary one can. For example, a series of MCI_SET commands could be made, specifying the following parameter sequences:

MCI_CDXA_AUDIO_DEVICE channel 0 MCI_SET_ON
MCI_CDXA_AUDIO_DEVICE channel 0 MCI_SET_OFF
MCI_CDXA_AUDIO_DEVICE channel 1 MCI_SET_ON

The MCI_SET commands would change the channel of the MCI_CDXA_AUDIO_DEVICE from 0 to 1.

XA Stream Connector

The *XA stream* connector represents the flow of XA ADPCM audio from the CD-XA device to its associated amplifier-mixer (ampmix) device. During playback, the CD-XA device sends audio information from a CD-ROM/XA disc to the ampmix device for subsequent conversion into audio that can be heard through conventional speakers or headphones.

Although volume control is provided by the CD-XA device, the device actually uses the services provided by the ampmix device to modify the perceived volume level. If other advanced audio-shaping features are required, the application can retrieve the device ID of the ampmix device using the MCI_CONNECTION message. Once the device ID has been obtained, the application can send set commands directly to the ampmix device to manipulate audio attributes, such as treble, bass, balance, and so forth.

Changing the Disc

If the user ejects a disc while the CD-XA device is playing, the device is stopped. When the CD-XA device detects that the disc has been changed and a file was opened on the previous disc, it closes the current CD-XA file and resets to the default state of the device. This state is identical to opening the CD-XA device without specifying a file. Following the reset, the CD-XA device returns MCIERR_MEDIA_CHANGED on the first command following detection of the disc change.

To avoid the interruption of play by the ejection of the disc, the application can disable a manual eject with an MCI_SET command before beginning playback.

CD-XA Command Messages

Message	Description
MCI_CLOSE	Closes the CD-XA device.
MCI_CONNECTOR	Enables or disables a connector, queries its state or identifies its type.
MCI_CUE	Cues the device for minimum delay in playback.
MCI_GETDEVCAPS	Gets device capabilities.
MCI_INFO	Fills a user-supplied buffer with the following information: -Product name and model number of the current audio device -Serial number (UPC) of the disc -Name of the currently loaded file
MCI_LOAD	Loads a data element into the CD-XA device.
MCI_OPEN	Initializes the CD-XA device.
MCI_PAUSE	Suspends the current playback action.
MCI_PLAY	Starts playing audio data from the device element. The following optional action modifies MCI_PLAY: -Specify start and stop positions in the device element
MCI_RESUME	Resumes playing from a paused state, keeping previously specified parameters in effect.
MCI_SEEK	Moves to the specified position in the device element.
MCI_SET	Sets audio attributes: -Identify the channels to be used -Set the volume -Enable or disable the audio output Retracts the tray and closes the door, if possible Opens the door and ejects the tray, if possible

	Sets the time format in milliseconds or MMTIME
MCI_SET_CUEPOINT	Sets a cue point.
MCI_SET_POSITION_ADVISE	Sets a position change notification request.
MCI_SET_SYNC_OFFSET	Sets a synchronization offset.
MCI_STATUS	Receives status on items such as the following: -Current volume setting -Whether the media is present in the device; -Current mode of the device, for example, "stopped" -Current position in the media -Destination of the data identified by channel number
MCI_STOP	Stops playing the CD-XA device.

Videodisc Device

The physical videodisc device is an external hardware device that plays videodiscs, producing an analog video and audio signal. The analog video output can be connected to an analog video digitizer component that produces output on the display, or it can simply be connected to a an external video monitor. The audio output can be connected to amplified speakers, headphones, or other audio media drivers.

Device Specifics

The videodisc media device supports Pioneer models 4200, 4300D, 4400, and 8000. The 4300D supports both the American (NTSC) and European (PAL) TV standards for storing analog video signals. Default communications settings for the players are: 8 data bits, no parity, 1 stop bit with a baud rate of 4800. The first communications port is also selected by default. These settings may be changed using the Multimedia Setup application if they do not match the current settings of the actual videodisc device. It is highly recommended that communication speeds at or above 4800 bits per second (bps) be utilized to ensure reliable and optimized performance.

Because the physical connections on the back of each videodisc device varies, the driver updates the connector and product information when the device is first opened. Applications can retrieve connector information, using MCI_CONNECTORINFO, and product information, using MCI_INFO.

Videodisc Formats

A videodisc has one of two formats: CAV or CLV.

CAV (constant angular velocity) is the interactive format of videodiscs, which allows freeze frame and slow motion. The CAV videodisc can be addressed by chapter or by frame. A CAV videodisc can contain up to 30 minutes of video on each side of the disc. One disc holds up to 54,000 frames.

With a CAV format videodisc, the angular velocity of the disc is always the same (1800 RPM), no matter what track is being read, which is similar to the way a phonograph record is played.

CLV (constant linear velocity) is the format for extended-play videodiscs. The CLV videodisc can be addressed by chapter or by time. A CLV videodisc can contain up to 60 minutes of video on each side of the disc.

With a CLV format videodisc, the linear velocity of the disc is constant under the laser head; the disc spins slower when the outside tracks are read.

Note: The number of chapters on CAV and CLV discs varies, depending on the manufacturer of the disc.

Using the Videodisc Device

Using the videodisc device consists of opening, configuring, seeking/stepping, playing, and setting cue points.

Opening the Device

The videodisc player is a dynamic single context device and therefore is serially shareable. When you open the videodisc player, it takes about 60 seconds for the disc to load and spin up to a playing position. Specifying a **notify** flag on the open request allows your application window procedure to remain available to process PM messages.

```
open videodisc alias video1 notify shareable
```

If you set the **shareable** flag, you should not start a playback operation until the system posts the asynchronous MM_MCIPASSDEVICE message with an event of MCI_GAINING_USE to your application.

When the system posts the asynchronous MM_MCINOTIFY message with a return code of MCI_NOTIFY_SUCCESSFUL message to your application, this indicates a device context is created. Although you may not have received the MCI_GAINING_USE event in a MM_MCIPASSDEVICE message, you can still make inquiries about the device and the media. The **status**, **capability**, **info**, and **close** commands can be sent to an inactive device context.

You may want to ensure that the disc in the player is not changed by the user. Some devices allow you to disable the manual eject. Other devices allow you to check the label on the disc:

```
info video1 label wait
```

Specify the wait flag with commands that return information. A string for the label is returned in the user-supplied buffer specified with mciSendString.

Use the **status** command to determine the current state of the videodisc player:

Query	**Response**
ready	TRUE or FALSE
mode	not ready, open, pause, park, play, scan, seek, or stop.
forward	TRUE, if player is set to play in forward direction.
time format	milliseconds, MMTIME, frames, chapters, HMS, HMSF.
position	Position in current format.
speed format	% or FPS.
speed	Speed in current format.

The **status** command also returns a lot of information about the disc:

```
status video1 media present wait
```

Query	**Response**
media present	TRUE or FALSE
media type	CAV, CLV, other
disc size	8 or 12
disc side	1 or 2
number of tracks	Number of tracks on the disc.

current track	Chapter number, if applicable.
length	Length of the current segment.
Start	Starting position of the media.

Use the **capability** command to query device capabilities for a particular format. You can request either CAV or CLV information. If no format is indicated, the default is CAV.

```
capability video1 clv can reverse wait
```

Configuring the Device

Before you begin playing the disc, you may want to set the audio channels. Most videodisc players have two channels for audio. On some videodisc players, the channels are used to produce a stereo effect. Other videodisc players allow you to turn one of the channels off, based on a selection by the user. This feature is useful for offering the user a choice of language, a level of instruction, and so forth.

Most videodisc players allow you to set the volume off (zero) or on (greater than zero) but do not offer a range of values for volume.

```
set video1 audio left off
```

You can set the speed format to frames-per-second or a percentage of the normal rate. The default for the speed format is frames-per-second, and the default for the time format is frames.

Most videodisc players have an on-screen display that you can set on or off. The display is a counter that keeps track of your position. The on-screen display is useful for debugging an application-for example, an interactive course. The display shows you where you are in relation to where you may want to be.

Seeking and Stepping

You can seek and step to a location on the disc using any of the videodisc time formats. Seeking is a fast-forward or fast-reverse to an absolute position. Stepping is done, backward and forward, in time units relative to the current position. The default for a step is one time unit forward.

```
seek video1 to start
step video1 by 1
```

Some videodisc players do not display a picture when you do a step. You get a "squelch" color, which is the color of the screen when a frame is not yet displayed. Some devices, particularly for CLV, cannot be frame accurate.

Playing a Videodisc

Some videodisc players can vary the playback speed of the CAV disc and also play the disc in reverse. Some players also can perform these operations on a CLV disc. Following are **capability** queries and the responses generated by a Pioneer 8000 device for CAV and CLV discs:

Query	**Response**
can reverse	TRUE
normal play rate	30 fps or 100%
fast play rate	90 fps or 300%
slow play rate	10 fps or 33.33%
maximum play rate	127 fps or 423.33%

minimum play rate	1 fps or 3.33%

Specifying a **seek** command before a **play** command can reduce the delay associated with a PLAY command.

A **play** command can specify a **to** and **from** position. If **from** is omitted, playing begins at the current position. If **to** is omitted, playing stops at the end of the disc, or at the beginning of the disc, if playing in reverse.

```
play video1 speed 30 fps
```

Note: Some videodisc players that cannot vary the playback speed return the same playback rate in response to these queries.

Setting Cue Points and Position Advises

A cue point is a location in the media that issues an MM_MCICUEPOINT notification message whenever it is encountered. The message is returned to the window specified when the cue point was set. Although a cue point is specified in the current time format with **setcuepoint**, the MM_MCICUEPOINT message is always returned in MMTIME units. MMTIME units are used because the time format set when the cue point is set and the time format set when the cue point is reached may be different. An application specific value can also be associated with a particular cue point for return in the MM_MCICUEPOINT message. The value can be anything which has meaning to the application. Generally, up to 20 cue points can be set.

```
set video1 time format frames
setcuepoint video1 on at 1500 return 1
setcuepoint video1 on at 4000 return 2
```

An MM_MCIPOSITIONCHANGE notification message is issued at periodic intervals as time elapses in the media for a particular device context. The message is returned to the window specified when the position advise was set with **setpositionadvise**. Only one position advise can be set for a device context. As with **setcuepoint**, the position specified is assumed to be in the currently selected time format while the position reported in the MM_MCIPOSITIONCHANGE notification is in MMTIME units.

For more information on cue points and position advises, see Cue Points and Position Advises.

```
setpositionadvise video1 on every 100
```

Because videodisc players are external devices and not part of the computer system, the IBM videodisc media driver can guarantee the accuracy of **setcuepoint** and **setpositionadvise** functions only within 10 frames, rather than the desired 100 milliseconds (3 frames) of the specified value.

Videodisc Player Error Return Values

The following table contains a list of error messages specific to videodisc players.

Return Code	Cause of the Error Message
MCIERR_VDP_COMMANDFAILURE	Videodisc players are external RS-232 devices; therefore, it is possible for a command to fail because of a device failure.
MCIERR_VDP_COMMANDCANCELLED	A command with the MCI_WAIT flag specified is either aborted or superseded.
MCIERR_VDP_NOSIDE	The videodisc player is unable to determine the side of the videodisc. This message can be returned by an MCI_STATUS MCI_VD_STATUS_SIDE request.

MCIERR_VDP_NOSIZE	The videodisc player is unable to determine the size of the videodisc. This message can be returned by an MCI_STATUS MCI_VD_STATUS_DISC_SIZE request.
MCIERR_VDP_INVALID_TIMEFORMAT	This message can be returned by any command specifying a chapter-specific parameter for a device that does not have chapters. It also can be returned if MCI_SET_SYNC_OFFSET is issued when the device is set to the chapter time format.
MCIERR_VDP_NOCHAPTER	Chapter information is not present on the disc. The following commands require chapter information to be present on the disc: MCI_SET_TIME_FORMAT MCI_FORMAT CHAPTERS, MCI_STATUS MCI_STATUS_CURRENT_TRACK, MCI_STATUS MCI_STATUS_NUMBER_OF_TRACKS.
MCIERR_VDP_NOTSPUNUP	The following commands require the videodisc to be spun up: MCI_INFO MCI_VD_INFO_LABEL and MCI_STATUS with the following parameters specified: MCI_STATUS_POSITION, MCI_STATUS_CURRENT_TRACK, MCI_STATUS_NUMBER_OF_TRACKS, MCI_STATUS_LENGTH, MCI_VD_MEDIA_TYPE, MCI_VD_STATUS_SIDE, MCI_VD_STATUS_SPEED, MCI_VD_STATUS_DISC_SIZE.

Videodisc Command Messages

Message	Description
MCI_OPEN	Initializes the videodisc device.
MCI_GETDEVCAPS	Gets device capabilities.
MCI_ESCAPE	Sends custom information to the media driver.
MCI_CUE	Cues the device for minimum delay in playback.
MCI_PLAY	Starts playing the videodisc. The following optional actions modify MCI_PLAY: -Specify start and stop positions -Play faster than normal -Play slower than normal -Play in reverse
MCI_PAUSE	Suspends the current playback action for CLV and CAV discs. Some players may also freeze the video frame.
MCI_RESUME	Resumes playing from a paused state, keeping previously specified parameters in effect.
MCI_SEEK	Searches, using fast forward or fast reverse, with video and audio off. The following optional actions modify MCI_SEEK: -Seek in reverse -Seek to the start or the end of the disc
MCI_SET	Sets audio attributes: -Identify the channels to be used -Set the volume

	-Enable or disable the audio output. Retracts the tray and closes the door, if possible Opens the door and ejects the tray, if possible Sets position format in frames, HMS, HMSF, milliseconds, MMTIME, or chapters Sets speed format in frames-per-second or as a percentage Disables or enables video output.
MCI_STATUS	Receives status on items such as the following: -Disc size -Which side of the disc is loaded -Whether current play direction is forward or backward -Length of the segment -Whether the media is present in the device -What the media type is: CAV, CLV, or other -What the current speed format is -Current mode of the device; for example, "stopped" -Current position in the media.
MCI_INFO	Fills a user-supplied buffer with the following information: -Product name of the device the peripheral is controlling.
MCI_STOP	Stops playing the videodisc device.
MCI_SET_CUEPOINT	Sets a cue point.
MCI_SET_POSITION_ADVISE	Sets a position change notification request.
MCI_SPIN	Starts or stop the disc from spinning.
MCI_STEP	Steps the play one or more time units forward or backward.
MCI_CLOSE	Closes the videodisc device.
MCI_CONNECTOR	Enables or disables a connector, query its state or identify its type.

Digital Video Device

OS/2 Version 2.1 or later supports playback of software motion videos. The ability to create and record movies is provided by the Video IN product (part of the OS/2 BonusPak). The OS/2 digital video device (**digitalvideo**) provides file-format support through the multimedia input/output (MMIO) architecture. Several movie file formats including AVI, MPEG, FLC/FLI animation are supported. Several decompression types, like Ultimotion, Indeo 2.1, 3.1, and 3.2, FLC/FLI animation, and MPEG-1 are also supported. Files containing interleaved video and audio are supported, as well as video-only files.

Compression Formats

The digital video device provides video playback and recording support through its open compressor/decompressor (CODEC) architecture.

The following table describes the digital video compressors and decompressors (CODECs) available with OS/2. The CODECs are represented by unique FOURCC identifiers. A FOURCC is a 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters (padded on the right with blank characters). For more information on CODEC procedures, see CODEC Procedures.

CODEC	FOURCC	Description
Ultimotion	ULTI	Compressor/Decompressor (Real-time and Asymmetric Compression)
Indeo 2.1	rt21/RT21	Compressor/Decompressor
Indeo 3.1	iv31/IV31	Compressor/Decompressor (Real-time and Asymmetric Compression)
Indeo 3.2	iv32/IV32	Decompressor
Uncompressed (RAW)	DIB	Decompressor
FLI/FLC	FLIC	Decompressor
MPEG-1	MPEG	Decompressor (RealMagic HW Decompression)

About Ultimotion

Advances in microprocessor power, data storage, and compression technology have provided key technologies for creating and playing digital video data on personal computers. The high-capacity disk drives and CD-ROMs satisfy the large storage needs of digital video data. Additionally, today's more powerful microprocessors provide sufficient power to handle digital video data in real time. When these advances are combined with image compression techniques, the result is a powerful integration of video and the personal computer.

Several compression algorithms are currently in use throughout the industry. Some of these algorithms, like MPEG, use additional video hardware to compress and decompress the digitized video. Others are less numerically intensive and can be handled by software running on the main CPU and still maintain sufficient frame rates to provide motion. These are referred to as *software-only* algorithms or *software motion video* .

Ultimotion is IBM's technology for software motion video. It is a cross-platform algorithm that uses no hardware acceleration for capture or playback. The following sections describe Ultimotion, a single compression technique capable of providing a spectrum of quality levels from a single copy of the digital video data:

- Data Stream Capabilities
  - Playback Characteristics
  - Resolution Scalability
  - Color Scalability
- Compression Ratios
  - Symmetric Compression
  - Asymmetric Compression
- Standard Ultimotion Movie

Data Stream Capabilities

Ultimotion is a playback-time scalable-video data stream. While the frame rate, output resolution, and color depth characteristics of a video are set when the video is created, the characteristics of a playback-scalable video are modified during playback utilizing the capabilities of the playback platform. These playback-platform capabilities depend on the type of microprocessor, display driver, video adapter, and data bandwidth available during playback.

Playback Characteristics

While Ultimotion is a playback-time scalable-video data stream, the magnitude that the data stream will "scale" is determined by the amount of information that is encoded in the data stream when it was created; that is, the amount of data placed in the data stream at creation time determines the "maximum" playback characteristics that a particular stream can achieve. In turn, the processing capabilities of the playback system determine how much of the data can be processed and presented during playback. Therefore, the playback characteristics of a given video are a function of the data put into the video by the author *and* the processing capabilities of the playback system.

The factors affecting the data stream at creation time are:

Resolution	Width and height of video
Frame duration	Frequency at which frames are to be created
I-frame rate	Frequency at which reference frames are to occur
Data rate	Average amount of data allowed for a second of video

Factors affecting the playback of a video are:

- Processing power of the CPU
- Throughput of data storage (for example, CD-ROM, hard disk, LAN)
- Efficiency of the display subsystem (such as the video adapter and display driver)

Resolution Scalability

Resolution of video determines how much spatial information is in a video file. Ultimotion compression algorithms organize this data so that it can be easily scaled up or down by factors of two as it is decompressed. Furthermore, as the data is decompressed, a sufficiently powered playback system can duplicate the data during output and display the video at four times its original size. This results in an effective output size larger than the input size. In this way, Ultimotion can be **scaled down** on systems incapable of processing the authored video resolution and **scaled up** on systems with more processing capability than the authored video requires.

Color Scalability

Ultimotion compression algorithms store images in 16-bit true color. This color is **scaled down** to the number of colors available on the playback system.

Compression Ratios

Ultimotion compressed frames use a coherent set of techniques for encoding a series of images. Different techniques are used for different purposes. Some represent detail very well while others represent large uniform areas with only a little data, and others fall in between but are very easily detected. Techniques can be mixed according to the needs of the image being compressed and are organized in an efficient manner for both compression and decompression. The result of mixing techniques is a data stream robust enough to be generated by software both asymmetrically and symmetrically.

Symmetric Compression

The Ultimotion compression techniques used on live video sources balance how well an image's detail is retained with how much time can be taken to analyze and compress the image. Symmetric Ultimotion compression uses the following items to determine the size and speed of symmetrically compressed Ultimotion videos:

- Image size
- Image quality (high, medium, low)
- Processor speed of the capture system

A 320 x 240 movie recorded at 15 frames per second at medium quality requires 9MB per minute of video. Reducing either the resolution, frame rate, or quality setting reduces the amount of storage required.

Asymmetric Compression

The Ultimotion compression techniques used during offline or asymmetric video recording take more time during the compression phase than it takes to decompress the video when the movie is played. By spending more time on compression, these techniques produce higher quality images and can compress the video data more effectively. Ultimotion asymmetric compression techniques produce movies stored using 320 x 240 resolution, 15 frame per second, and can be played from a CD. Movies made with these specifications require 9MB per minute. Reducing either the resolution, frame rate, or quality setting reduces the amount of storage required. Increasing these specifications require faster storage devices (for example, double speed CD-ROM or hard disk) and faster playback systems.

Standard Ultimotion Movie

Ultimotion compression algorithms use byte-oriented data structures and efficient data organization to provide software-only decompression. Since frame rate, resolution, and data rate are set when a movie is created, these settings determine the minimum platform required for playback. The "standard Ultimotion movie" is defined as 320 x 240, 15 frames per second at a 150KB per second data rate. This movie can be played on at least a 25 MHz 80386 microprocessor and an SVGA display adapter. Computers with 33 MHz 80486 microprocessors are capable of displaying 320 x 240 resolution at 24 frames per second or 640 x 480 resolution at 15 frames per second. The frame rate increases as the the data rate of the source device and the machine speed increases. A frame rate of 30 frames per second is possible on a machine with a 66 MHz 80486 microprocessor.

Playback and Recording

OS/2 multimedia provides support to OS/2 Presentation Manager applications for controlling playback and recording of motion video. Playback of video can be performed in either a default window or an application window. Recording can be performed using video capture hardware devices with appropriate device driver support. The digital video device can also be used to capture still image (bitmap) data from video capture hardware or previously recorded motion video files.

This section describes digital video functions and provides examples using media control interface string commands.

Digital Video Windows

The digital video device provides two methods of displaying the video for a movie: a *default window* and an *application-defined window* .

Attention: To avoid unpredictable results, it is important that a Workplace Shell application does not create a PM window for digital video on the main thread. The application must start a separate thread, create the window on that thread, and then create a new message queue on that thread to service the window.

Default Window

The default window is used if no other window handle is specified with the **window** command. The parent of the default video window is HWND_DESKTOP unless otherwise specified with the **open** command. It provides a basic desktop window in which to display video that can be moved, sized, and minimized by the user. The default window is sized to the size of the video. The digital video device provides and manages this window for the application.

The default window has **Normal Size**, **Double Size**, and **Half Size** options on its settings menu. Selecting these options cause the window to be resized. The window can also be resized to other sizes, including icon size.

Application-Defined Window

An application-defined window can be used when the application requires more control over the window. The application can place video in its own client area or in a child window, add menus, and so on. If an application specifies a parent window handle when opening the device, it must close the logical video device before destroying the parent window.

If an application passes a window handle to the digital video device with MCI_WINDOW, it is essential that the application ensure that this window will receive all WM_REALIZEPALETTE messages sent to the message queue. If the window is a client-frame window (its parent has a style of CS_FRAME and the window has an ID of FID_CLIENT) or the child of a client window, it will receive this message automatically. If, however, the window has an ancestor that is not a client window and does not pass WM_REALIZEPALETTE to WinDefWindowProc, the window will not automatically receive the message. For example, WinDefDlgProc does not pass WM_REALIZEPALETTE on to child windows. So, if an ancestor of the window is a dialog window, the dialog procedure must explicitly pass the WM_REALIZEPALETTE message to the window. If the window does not receive this message, then incorrect colors will appear in video displayed in the window, whenever another application changes the system palette.

When an application-defined window is used to display video, the digital video device subclasses the window to ensure that video updating is maintained correctly. Subclassing the window does the following:

- Positions and sizes the window
- Prevents the window from being sized above the maximum supported size
- Modifies the tracking rectangle so the user cannot drag the size border beyond the screen boundaries

----

Opening the Device

The MCI_OPEN message is issued to create a device context of the digital video device. A default window is created and displayed when the device is opened if the device is cued for output or a movie is loaded. The video device directs its output to the default window. If a window is specified with the MCI_WINDOW message the video device directs its output to the window specified. For the convenience of applications that are using the string interface, some window-style control functions are also exported as flags to the MCI_WINDOW message.

Whether the default window or an application-defined window is used for output, the digital video device confines its output to the device coordinates of the specified window.

The default video window is invisible when the device is first opened. This allows the user or application to prepare the size, position, and contents of the window before it is displayed. It is created in a frame window, which can be sized, moved, maximized, and minimized. It is also created with an ideal aspect ratio in the center of the display and occupies approximately one quarter of the screen. As the user changes the size of the window, the digital video device scales the video image as required to maintain the video within the current coordinates. Maintaining a constant aspect ratio is the responsibility of the application program.

The parent of the default video window is HWND_DESKTOP unless otherwise specified in the MCI_OPEN message. The owner of the default video window is NULL. An application can specify a parent window handle for the default video window using *hwndParent* when the device is opened. This is the only window-related parameter that can be specified using media control interface commands that changes the behavior of the default video window.

Playing Motion Video Files

By default, playback of digital motion video is displayed in the window supplied by the digital video device. This window is created and displayed when the video is cued for output.

The following string commands illustrate playing an entire AVI format file. Because **to** and **from** flags are not specified, the file is played from the current position to the end of the file. When a motion video device element is opened, the current position in the media is the first playable area after any header or table of contents information.

```
open movie.avi type digitalvideo alias myvideo wait
play myvideo notify
close myvideo
```

Each frame in a motion video file has a number associated with it. From the perspective of the digital video device, each file is zero-based. That is, the first frame is frame 0, the second frame is frame 1, and so forth. This means the number of the last frame in a file is 1 less than the total number of frames in the file.

The current position always indicates the frame that is *about* to be displayed rather than the frame that is currently displayed.

When a play position is specified with the **from** flag, the actual position reached is accurate only to the nearest intracoded frame (I-frame). However, a position specified with the **to** flag is exact.

If you need to specify an exact position in the video file to play from, you can issue the **seek** command, which moves the current position in a file to an exact point. The following string commands illustrate moving the current position to frame 20 and then playing to frame 100.

```
open movie.avi type digitalvideo alias myvideo
set myvideo time format frames wait
seek myvideo to 20 wait
play myvideo to 100 notify
close myvideo
```

The **cue** command can also be used to seek and cue a particular frame. By specifying the **show** or **noshow** flag, you can control whether the video window will be displayed or hidden when the cue operation is performed. This can be useful for displaying video frames as a user moves the position slider, providing visual feedback of the location in the video a user is seeking to.

```
cue myvideo show to 20 wait
```

If a **to** position is *not* specified, the current frame is displayed and the media position will advance by one (frame).

Playing Hardware-Assisted MPEG Files

The RealMagic adapter enables hardware-assisted playback of MPEG files. MPEG video content is contained in an MPEG-specific file format. This file format, like AVI, interleaves audio and video data but is quite different from the tagged AVI format and requires a unique IOProc. The MPEG IOProc performs the file-format processing in the hardware-assisted environment, while decompression of both audio and video is performed by the hardware.

The digital video device provides normal-speed playback support (in the forward direction only) of MPEG files. The common file extension for MPEG files is .MPG. Seeking is supported, however seeking is an approximation to the nearest MPEG *picture group* . The size of a picture group is movie-dependent but is commonly 8 frames, so seeking takes place to within one-third of a second, in most cases.

Playing Animation Files

In addition to playing motion video with the digital video device, you can also play FLC and FLI animation files. FLC and FLI are the standard file formats for most animation tools. The FLI file format is most common. FLI is limited to a 320 x 200 display resolution and a custom 256-color palette. The FLC file format is an extension to FLI and allows a 640 x 480 resolution.

```
open cartoon.flc type digitalvideo alias animate
play animate notify
close animate
```

The FLC/FLI video files do not contain any audio data. However, the digital video device will search for an audio file (.WAV) with the same base name as the animation file in the same directory. If such a file exists, the audio is played with the animation file In the previous example, if a file named CARTOON.WAV exists, the audio file plays with the animation file and continues to play until CARTOON.FLC ends.

Seeking within a FLC/FLI file is not supported because these files contain only one initial I-frame. All other frames in the file are delta frames. All seek requests result in a seek to the beginning of the file.

Recording Motion Video

The digital video device supports real-time recording of motion video into AVI files. The supported compression algorithms for recording video are Ultimotion, Indeo 2.1, and Indeo 3.1. The default settings for real-time recording are 160 x 120 resolution, 15 frames per second, and Ultimotion compression type.

Recording into new and existing files is supported. The following example illustrates recording live video. As recording takes place, the digital audio and video data is stored in the temporary file created for the video device element. After the recording operation is complete, the device element is played back so it can be viewed before it is saved as a permanent file on disk.

To save the data as a video file, you can specify an existing file name or a new file name with the **save** command. If you indicate an existing file name, the data in the disk file is completely replaced by the data in the temporary file. You can also indicate that the file being saved is a *video* file; however, it is not necessary because this is the default.

In the example, monitoring is set on, so that the incoming video signal can be viewed in the default video window before it is recorded. Monitoring of live video can also be done without recording.

```
open digitalvideo alias myvideo wait
set myvideo time format frames wait
set myvideo monitor on wait
record myvideo to 99 wait
play myvideo wait
save myvideo newvid.avi video wait
close myvideo
```

Applications can specify that only video or only audio is to be recorded. In the following example, audio recording is turned off, so that only video will be recorded.

Specifying the **cue input** command causes the default window to become visible, so it is not necessary to set the monitoring function on. The **cue** command ensures the device is initialized and ready to record, preventing the possibility of losing any initial video when recording begins. As in the previous example, the device element is played back before it is saved as a file on disk.

```
open digitalvideo alias myvideo wait
set myvideo record audio off wait
set myvideo time format hms wait
cue myvideo input wait
record myvideo to 00:02:00 wait
play myvideo wait
save myvideo newvid.avi wait
close myvideo
```

Image Support

The digital video device provides image support to applications that have the following requirements for digital video:

- Display an image of the current frame when the movie is stopped.

- Create and save an image in the specified format.

- Provide an image buffer to drag and drop cut-and-paste objects.

Capturing a Still Image

There are two functions applications can use to capture an image:

- MCI_GETIMAGEBUFFER
- MCI_CAPTURE

An application uses MCI_GETIMAGEBUFFER to capture an image when it wants to keep the image in an application buffer. The application can capture the image from either a movie file, or the buffer of a video capture card. This command message returns the image in the format specified by the application.

An application uses MCI_CAPTURE to capture an image from a movie file when it wants the image kept in a digital video device element.

Editing Operations

The digital video device supports the use of application buffers by editing commands to support drag and drop functions. The following flags are supported by the editing commands:

Message	Flags
MCI_CUT	MCI_TO_BUFFER
MCI_COPY	MCI_FROM_BUFFER, MCI_TO_BUFFER
MCI_PASTE	MCI_FROM_BUFFER, MCI_TO_BUFFER

To determine the buffer length for an editing operation, the application can issue MCI_CUT or MCI_COPY with either the MCI_FROM_BUFFER or MCI_TO_BUFFER flag set and a zero buffer length. The editing operation will not be performed, but the length of the buffer that is necessary for the operation will be returned in the buffer length field.

Setting the Tuner Device

With the appropriate hardware, the digital video device is able to monitor a TV channel. The WinTV card is an overlay card that has video capture support as well as television signal support. The monitor window displays the TV signal. The tuner is selected by setting the digital video device connector type to **video in** and the connector number to the tuner connector number of the adapter. The tuner connector number varies from adapter to adapter. On the WinTV adapter, this is connector number 2.

```
connector digitalvideo02 enable type video in number 2 wait
```

An application can use the **settuner** command to set the region, channel, and fine-tuning values. These flags are valid when used individually or in any combination. The digital video device uses these three values to calculate a frequency to send to the device. See the following example.

```
open digitalvideo03 alias mytuner wait
settuner mytuner tv channel 3 region usa finetune plus 1 wait
```

The following example illustrates setting the frequency directly. Setting the frequency directly causes channel, region, and fine-tuning values to be ignored.

```
settuner mytuner frequency 80 wait
```

The region value must have a corresponding ASCII file (*region* .RGN) located in the \MMOS2\REGION subdirectory. When an application selects the region with the **settuner** command, the digital video device reads in the region file and fills a region array. The channel is used as an index into this array, which identifies the frequency. Fine-tuning is then added to the frequency. Region files are keyword-driven and have the following format:

```
[tuner]
  description=USA Cable
  lowchannel=2
  highchannel=99
  frequencies=0,0, 10050, -1, ... 29000
```

The *description* keyword identifies a descriptive string displayed on the corresponding Multimedia Setup page. The *lowchannel* and *highchannel* keywords indicate the range of channels available for the region. The frequencies correspond to the channel values, beginning with channel 0 and continuing to the highest available channel. The frequencies must be listed in columns of nine; otherwise, the region is invalid. A frequency value of 0 indicates the channel is out of range. A frequency value of -1 indicates a skipped channel, allowing the user to "block" certain channels from being viewed.

Digital Video Command Messages

Message	Description
MCI_CAPTURE	Captures the current video image and stores it as an image device element.
MCI_CLOSE	Closes the device element and any resources associated with it.
MCI_CONNECTOR	Enables, disables, or queries the status of connectors on a device.
MCI_COPY	Copies specified data range into clipboard or buffer.
MCI_CUE	Prompts the device to ready itself for a subsequent playback or recording operation with minimum delay.
MCI_CUT	Removes specified data range and places it into clipboard or buffer.
MCI_DELETE	Deletes specified data range. Clipboard or buffer is not used.
MCI_GETDEVCAPS	Gets device capabilities.
MCI_GETIMAGEBUFFER	Retrieves the contents of the capture video buffer or the current movie frame.
MCI_INFO	Returns string information from the device.
MCI_LOAD	Specifies a new file to be loaded onto an already existing device element.
MCI_OPEN	Initializes the device.
MCI_PASTE	Deletes selected data range if difference between FROM and TO is more than zero, then inserts data from buffer or clipboard.

MCI_PAUSE	Pauses playback or recording.
MCI_PLAY	Starts a play operation on the device.
MCI_PUT	Defines the source and destination rectangles windows.
MCI_RECORD	Starts a recording operation on the device.
MCI_REDO	Reverses previous MCI_UNDO command.
MCI_RESTORE	Transfers an image from the element buffer to the display surface.
MCI_RESUME	Resumes playing or recording from a paused state, keeping previously specified parameters in effect.
MCI_REWIND	Rewinds or seeks the device element to the first playable position (beginning).
MCI_SAVE	Saves the current file to disk.
MCI_SEEK	Moves to the specified position in the file.
MCI_SET	Sets device information.
MCI_SET_CUEPOINT	Sets a cue point.
MCI_SET_POSITION_ADVISE	Sets a position change notification for the device.
MCI_SETTUNER	Sets the frequency for the tuner device.
MCI_STATUS	Obtains status information for the device.
MCI_STEP	Steps the playback one or more frames forward or backward.
MCI_STOP	Stops playing or recording.
MCI_UNDO	Cancels previous RECORD, CUT, PASTE, or DELETE.
MCI_WHERE	Obtains a rectangle array, specifying the source or destination area.
MCI_WINDOW	Specifies the window and the window characteristics that the device should use for display.

Direct Interface Video Extensions (DIVE)

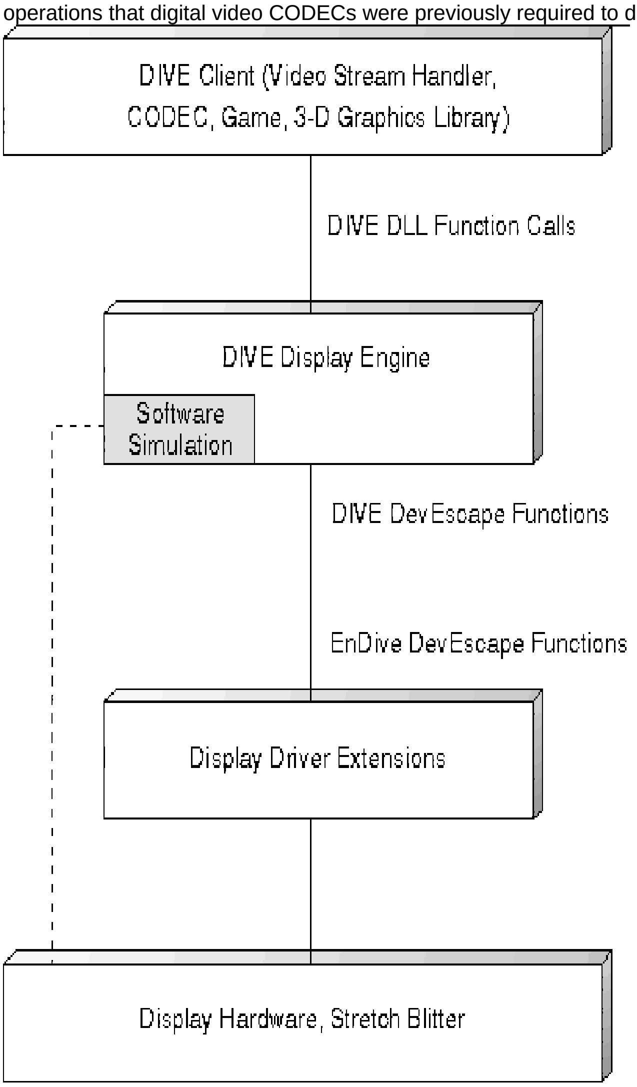
Software motion video implementation under OS/2 has attained sizable performance advantages by enabling video decompressors to directly write to video memory. While this technique provides good performance, it has the disadvantage that each decompressor must deal with the pel format of the display in various modes, clipping the output to visible regions, and any scaling that is to be performed. Additionally, on bank-switched video displays, the decompressor must return on partial frames to enable the video stream handler to switch banks. The direct interface video extensions (DIVE) consolidate the complexities of dealing with direct video frame buffer access (sometimes referred to as "direct access" or "black hole") into a single API DLL that enables efficient transfer to video memory with clipping, scaling, and color space conversion. The optimized screen access functionality provided by DIVE can be used for motion video display, fast image updates for interactive games, and fast screen display by 3-D graphics libraries.

About DIVE

DIVE is a DLL that provides optimized blitting performance for motion video subsystems and applications that perform rapid screen updates in the OS/2 PM and full-screen environments. Using DIVE functions, applications can either write directly to video memory or use the DIVE blitter to get a high level of screen update performance, image scaling, color space conversion, and bank-switch display support. The DIVE blitter utilizes acceleration hardware when present and applicable to the function being performed.

DIVE Display Engine Functional Characteristics

The DIVE system-level interface abstracts the low-level device driver DIVE interface to a higher level and adds software emulation for operations that digital video CODECs were previously required to do. This system-level interface is referred to as the *DIVE display engine* .


DIVE Client (Video Stream Handler,
CODEC, Game, 3-D Graphics Library)
DIVE DLL Function Calls
DIVE Display Engine
Software Simulation
DIVE DevEscape Functions
EnDive DevEscape Functions
Display Driver Extensions
Display Hardware, Stretch Blitter

The DIVE display engine consists of a single, stand-alone DLL that exports the following functions:

- Query for display capabilities
- Open instance
- Visible region specification
- Allocation of image data buffers
- Buffer-to-screen/buffer-to-buffer transfer (blitter) setup
- 8-bit CLUT color palette simulation/specification
- Transfer buffer to image display/transfer buffer to buffer
- Utility functions useful for direct access, including window starting-address calculation
- Frame buffer acquire/deacquire
- VRAM bank selection (for bank-switched displays)

- Close instance

The display engine enables subsystem components (for example, video CODECs) and applications to either directly access the video buffer or to use the display engine screen transfer functions. If direct access is used, the using component or application is responsible for writing to the frame buffer format for the current display mode and correctly clipping the output. In addition, the component is responsible for acquiring and bank switching the display apertures. If display engine screen transfer functions are used, the display engine handles clipping, pel format and color space conversions, and any necessary hardware serialization. The input formats and their corresponding color encoding specification values are:

- CLUT 8 (256 color) - "LUT8"
- 8-bit greyscale - "GREY"
- RGB 16 (5-6-5, 5-5-5) - "R565", "R555", "R664"
- RGB 24 (R-G-B, B-G-R) - "RGB3", "BGR3"
- RGB 32 (R-G-B, B-G-R) - "RGB4", "BGR4"
- YUV 9 - DVI/Indeo three-plane color subsampled - "YUV9"
- YUV 422 - "Y422"
- YUV CCIR601 - three-plane 2x2 color subsampled (MJPEG, MPEG) - "Y2X2"
- YUV CCIR601 - three-plane 4x4 color subsampled - "Y4X4"

The output formats are:

- CLUT 8 (256 color)
- RGB 16 (5-6-5, 5-5-5, 6-6-4)
- RGB 24 (R-G-B, B-G-R)
- RGB 32 (R-G-B-x, B-G-R-x)
- YUV 422 (Y-U-Y-V)

Using Dive

There are two main ways to use DIVE: using the DIVE blitter and using direct frame-buffer access. DIVE applications gain access to DIVE functions by obtaining a DIVE handle:

```
ULONG       ulErrorCode;
HDIVE       *phDiveInst;
BOOL        fNonScreenInstance;
PVOID       ppFrameBuffer;

ulErrorCode = DiveOpen( *phDiveInst, fNonScreenInstance, ppFrameBuffer );
```

A corresponding DiveClose function must be called at application termination. If DIVE is to be used for blitting to the screen, set *fNonScreenInstance* to FALSE. Otherwise, if DIVE is to be used only for off-screen sizing or color format conversion, set *fNonScreenInstance* to TRUE. If *fNonScreenInstance* is FALSE, a pointer to the frame buffer is returned in *ppFrameBuffer* .

DIVE Image Buffers

Because DIVE will use off-screen VRAM where available for acceleration of blitting operations, the application should allocate all source blitting buffers from DIVE whenever possible. To allocate a buffer, the application would make the following call:

```
ULONG   ulBufNum;
FOURCC  fccColorSpace;
ULONG   ulWidth, ulHeight, ulLineSizeBytes;
PBYTE   pbImageBuffer;

ulErrorCode = DiveAllocImageBuffer(
        hDive,          /* DIVE handle  */
        &ulBufNum,     /* Buffer number (output)       */
        fccColorSpace,  /* Color format */
        ulWidth, ulHeight,     /* Size of maximum image        */
        ulLineSizeBytes,
        &pbImageBuffer);
```

A corresponding DiveFreeImageBuffer function call is used to deallocate the buffer when it is no longer needed. The color format of the image buffer is described by *fccColorSpace* . The DIVE interface defines constants for a variety of 8-, 16-, and 24-bit color encoding schemes. After a buffer is allocated and before it can be used for blitting, it must be accessed as shown in the following example:

```
PBYTE   pbImageBuffer;
ULONG   ulBufferScanLineBytes, ulBufferScanLines;

ulErrorCode = DiveBeginImageBufferAccess(
        hDiveInst,          /* DIVE handle   */
        ulBufferNumber,       /* Buffer number */
        &pbImageBuffer,    /* Ptr to image buffer (output) */
        &ulBufferScanLineBytes);  /* Scan line length (output)    */
        &ulBufferScanLines);  /* Scan lines (output)    */
```

DIVE calculates the number of bytes per scan line for the image buffer (based on the color format) and returns the value in *ulBufferScanLineBytes* . The application can now write color data into *pbImageBuffer* . For example, the application could open a bit-map file and read the bit-map data directly into the image buffer. After the data has been written, the application calls DiveEndImageBufferAccess to deaccess the buffer. Be sure to use scan line bytes (you might have to read a line at a time).

DIVE Palettes

Applications must inform DIVE of the state of the physical palette upon initialization and whenever the physical palette changes. At application initialization, and in response to a WM_REALIZEPALETTE message, the application calls the following sequence:

```
BYTE       pbPal[1024];

/* Query the physical palette from PM   */
GpiQueryRealColors( hps, 0, 0, 256, (PLONG)pbPal);

/* Pass it to DIVE              */
DiveSetDestinationPalette( hDive, (PBYTE)pbPal);
```

If the application itself is using palettes, these palettes must also be communicated to DIVE through the DiveSetSourcePalette function. For example, if the application is using DIVE to blit 256-color palettized images to the screen, the application must send the image palette with a call to DiveSetSourcePalette. If a sequence of images is being used for animation and the palette remains constant through the series, it is necessary to call DiveSetSourcePalette only once before blitting the first image in the series.

DIVE provides high-speed screen updates by bypassing PM. In order to maintain the integrity of the PM desktop, DIVE applications must notify DIVE whenever the visible region of the application window changes so that output may be clipped accordingly.

Every DIVE application will request visible region notification at window initialization time with the following call:

```
WinSetVisibleRegionNotify( hwnd, TRUE);
```

The first parameter is the handle of the window where the graphics operations will appear, and the second parameter turns on notification for that window. (A corresponding WinSetVisibleRegionNotify(hwnd, FALSE) call should be made to turn notification off at window termination time.)

Whenever the window's visible region begins to change, either because the window is being moved or sized or another window or icon overlaying the window is being moved or sized, the window will receive a WM_VRNDISABLED message. In response to this message, the DIVE application will call DiveSetupBlitter (hDiveInst, 0). Once the movement is complete, the window will receive a WM_VRNENABLED message. In response to this message, the DIVE application will query the new visible region, using WinQueryVisibleRegion as follows:

```
hps = WinGetPS( hwnd );
hrgn = GpiCreateRegion( hps, 0, NULL);
WinQueryVisibleRegion( hwnd, hrgn);
```

Whenever the visible region, source color format, or image source or destination size changes, the DIVE application must pass the changes to DIVE with a call to DiveSetupBlitter. Note that it is necessary to pass the rectangles for the region in window coordinates and the position of the window in desktop coordinates. First, get the rectangles and window coordinates:

```
RECTL   rctls[50];      /* Rectangles for visible rgn   */
RGNRECT rgnCtl;         /* Region control struct        */
SETUP_BLITTER   SetupBlitter;   /* DiveSetupBlitter struct      */
POINTL  pointl;
SWP     swp;
HPS     hps;
HRGN    hrgn;

rgnCtl.ircStart = 0;    /* Enumerate rectangles */
rgnCtl.crc = 50;        /* Starting with first  */
rgnCtl.ulDirection = RECTDIR_LFRT_TOPBOT;

/* Get rectangles for the visible region        */
GpiQueryRegionRects( hps, hrgn, NULL, &rgnCtl, rctls);

/* Find the window position relative to its parent.     */
WinQueryWindowPos( hwnd, &swp );

/* Map window position to the desktop.  */
pointl.x = swp.x;
pointl.y = swp.y;
WinMapWindowPoints( WinQueryWindow( hwnd, QW_PARENT ),
        HWND_DESKTOP, &pointl, 1);
```

Then, pass the information to DIVE:

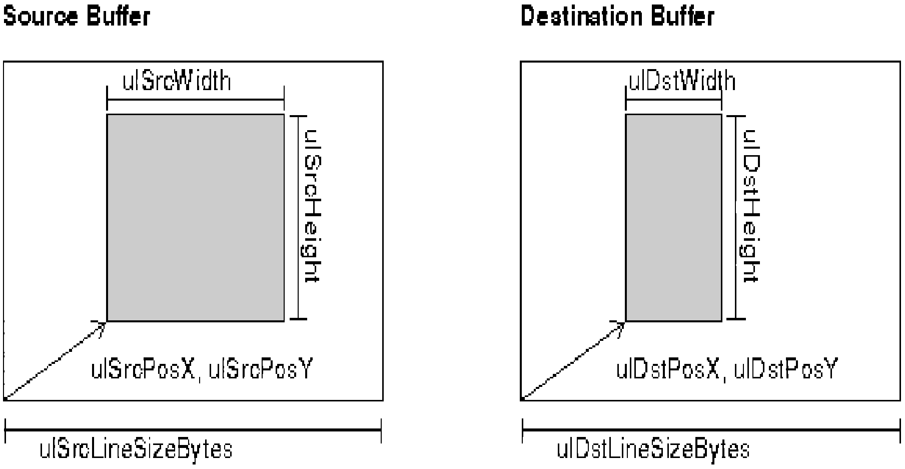
```
/* Tell DIVE about the new settings.  */
SIZEL   sizlSrcImg;     /* Size of source image */
FOURCC  fccSrcColors;   /* Source image format  */

SetupBlitter.ulStructLen = sizeof ( SETUP_BLITTER );
SetupBlitter.fInvert = 0;
SetupBlitter.fccSrcColorFormat = fccSrcColors;
SetupBlitter.ulSrcLineSizeBytes = ulScanLineBytes;
SetupBlitter.ulSrcWidth = sizlSrcImg.cx;
SetupBlitter.ulSrcHeight = sizlSrcImg.cy;
SetupBlitter.ulSrcPosX = 0;
SetupBlitter.ulSrcPosY = 0;
SetupBlitter.fccDstColorFormat = FOURCC_SCRN;
SetupBlitter.ulDstLineSizeBytes = 0;
SetupBlitter.ulDstWidth = swp.cx;
SetupBlitter.ulDstHeight = swp.cy;
SetupBlitter.ulDstPosX = 0;
SetupBlitter.ulDstPosY = 0;
SetupBlitter.lScreenPosX = pointl.x;
SetupBlitter.lScreenPosY = pointl.y;
SetupBlitter.ulNumDstRects = rgnCtl.crcReturned;
SetupBlitter.pVisDstRects = rctls;
DiveSetupBlitter ( hDive, &SetupBlitter );
```

The color format of the source image is described by *fccSrcColors* .

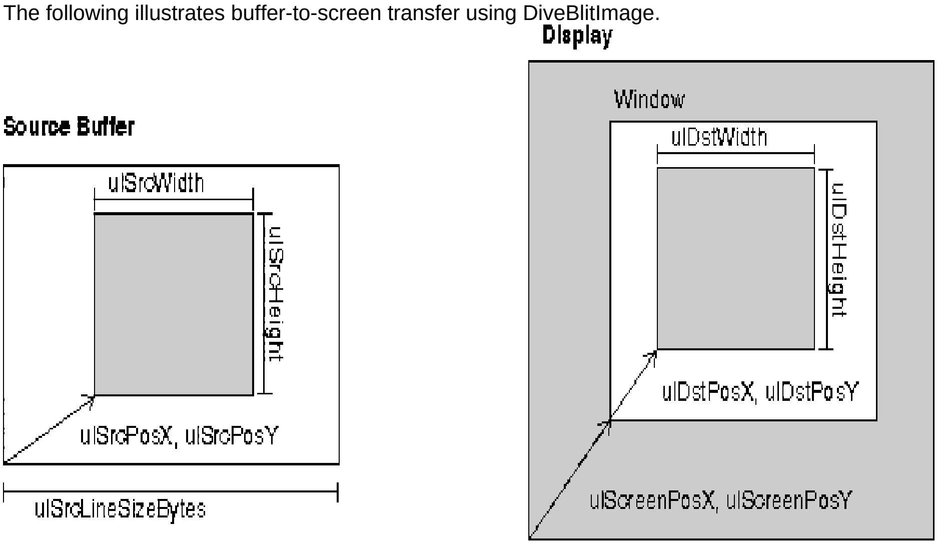
Note that, in this example, the DIVE blitter is set up to blit to the screen, but that need not be the case. DIVE could also be used to perform color conversion and/or stretch blitting to a destination image. The destination color-encoding format would be indicated in *fccDstColorFormat* ; *ulDstWidth* and *ulDstHeight* would be set to the size of the destination image; *ulNumDstRects* would be set to 1; and *pVisDstRects* would point to a single rectangle with *xLeft* =*yBottom* =0 *xRight* =*ulDstWidth* and *yTop* =*ulDstHeight* .

Blitter Operation

The following illustrates buffer-to-buffer transfer using DiveBlitImage.



The following illustrates buffer-to-screen transfer using DiveBlitImage.



Note: The screen cannot be used as a source for blitting using DIVE.

Transparent Blitting

DIVE transparent blitting functions enable an interactive game or imaging application to create composites of graphics and image data using a transparency key color.

To set up for transparent blitting, the application calls DiveSetTransparentBlitMode before it calls DiveSetupBlitter.

```
ULONG APIENTRY DiveSetTransparentBlitMode (HDIVE hDiveInst,
                                           ULONG ulTransBlitMode,
                                           ULONG ulValue1,
                                           ULONG ulValue2);
```

The following transparent blitting modes can be specified for *ulTransBlitMode* :

- DIVE_TBM_NONE
- DIVE_TBM_EXCLUDE_SOURCE_VALUE
- DIVE_TBM_EXCLUDE_SOURCE_RGB_RANGE

- DIVE_TBM_INCLUDE_SOURCE_RGB_RANGE
- DIVE_TBM_EXCLUDE_SOURCE_YUV_RANGE
- DIVE_TBM_INCLUDE_SOURCE_YUV_RANGE

Transparent blitting functions are based on source pixel values. If the pixel value in the source image buffer is the key color, the value of the corresponding pixel in the destination buffer is not modified. Interpretation of color values specified in *ulValue1* and *ulValue2* parameters is dependent on the source image color format specified in the *fccSrcColorFormat* field of the SetupBlitter structure and the transparent blitting mode.

Direct Frame-Buffer Access

As mentioned earlier, **ppFrameBuffer* returned by DiveOpen gives direct addressability to the frame buffer. In order to write directly to the frame buffer, the DIVE application must perform its own clipping, color conversion, and bank switching. The following functions are provided for direct frame-buffer access: DiveAcquireFrameBuffer, DiveDeacquireFrameBuffer, DiveSwitchBank, and DiveCalcFrameBufferAddress.

The DiveQueryCaps function returns whether the display subsystem is bank-switched. DIVE provides another function called DiveCalcFrameBufferAddress to get to a location in the frame buffer that corresponds to a rectangle in desktop coordinates:

```
PRECTL prectlDest;            /* Image rectangle in desktop coors  */
PVOID pDestinationAddress;    /* Frame buffer address - output     */
PULONG pulBankNumber;         /* Display h/w bank number - output  */
PULONG pulRemlinesInBank;     /* Lines left in bank - output       */

ulErrorCode = DiveCalcFrameBufferAddress(
        hDiveInst, &prectlDest, &pDestinationAddress,
        &pulBankNumber, &pulRemlinesInBank);
```

To accomplish correct clipping, *prectlDest* must be in the application window's visible region. If the display hardware is bank-switched, then the application must not write more than *pulRemlinesInBank* lines of output before switching banks. The data written to *pDestinationAddress* must be in the color-encoding scheme of the screen (also provided by DiveQueryCaps). (Of course, DIVE can be used to convert to the correct screen color-encoding prior to writing to the frame buffer by doing a DiveBlitImage to a destination buffer with the same color-encoding.) Additionally, if the screen supports only 256 colors, the data must match the current physical palette.

All write access to the frame buffer must be bracketed by calls to DiveAcquireFrameBuffer and DiveDeacquireFrameBuffer. Also, the application must not attempt to access the frame buffer between receipt of a WM_VRNDISABLED message and a WM_VRNENABLED message.

A typical application would do the following:

```
BOOL  fKeepBlitting = TRUE;
BOOL  fFBAccessOK;
RECTL rectlOutput;      /* Image rectangle in desktop coors  */
RECTL rectlDest;        /* Portion to blit in this bank */
ULONG ulMoreLines;      /* Lines in image left to blit  */
LONG  lBlitTop;         /* Top of next blit     */
PVOID pDestinationAddress;   /* Frame buffer address - output */
ULONG ulBankNumber;          /* Display h/w bank number - output */
ULONG ulRemlinesInAperature; /* Lines left in bank - output  */
BOOL  fAcquired = FALSE;     /* Acquired frame buffer yet    */

while (fKeepBlitting)
  {
  /* ... Call DiveSetupBlitter as before ...    */

  /******************************************************/
  /* Calculate total number of lines to blit.  Then blit  */
  /* only those lines that will fit in the current bank.  */
  /* Switch to successive banks until the entire image is */
  /* blitted.                                             */
  /******************************************************/
  ulMoreLines = rectlDest.yTop - rectlDest.yBottom;
  memcpy( &rectlDest, &rectlOutput, sizeof(RECTL));
  while (ulMoreLines && fFBAccessOK)
    {
    ulErrorCode = DiveCalcFrameBufferAddress(
        hDive, rectlDest, &pDestinationAddress,
        &ulBankNumber, &ulRemlinesInAperture);
    if (!fAcquired)
      if (!DiveAcquireFrameBuffer( hDive, ulBankNumber))
        fAcquired = TRUE;
      else break;
    DiveSwitchBank( hDive, ulBankNumber);
      {
      /* ... write data for (ulRemlinesInAperture) top lines of */
      /* rectlDest to pDestinationAddress ...   */
      if (ulRemlinesInAperture < ulMoreLines)
        {               /* if need next bank    */
        rectlDest.yTop -= ulRemlinesInAperture;
        ulMoreLines -= ulRemlinesInAperture;
        }
      else ulMoreLines = 0;
      }
    if (fAcquired)
      DiveDeacquireFrameBuffer( hDive);
    }           /* end: while more lines to blit */
  }             /* end: blitter loop    */
```

Note: This method works only on even bank breaks; indirect access through DiveBlitImage is recommended on displays with bank breaks that are not aligned on scan-line boundaries.

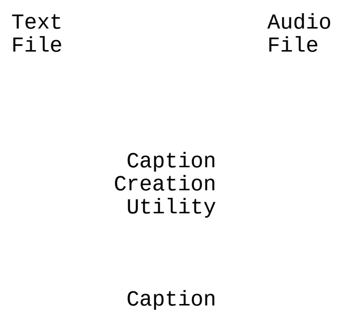
In the previous example, the application spins off a separate thread to perform blitting. The procedure for this thread contains a nested loop that switches display banks for each image blitted as long as blitting is needed. The flag fFBAccessOK is turned off whenever the application window received WM_VRNDISABLED and is turned on whenever WM_VRNENABLED is received.

Captioning

The Toolkit provides a sample captioning system to assist application programmers in adding captioning to multimedia applications. The sample captioning system consists of three parts: the Caption Creation Utility program, Caption DLL, and Caption Sample Application. The Caption Creation Utility program (located in \TOOLKIT\SAMPLES\MM\CAPTION subdirectory) creates a "caption" file. This is a text file containing timing information relating to its associated audio file. The Caption DLL (located in the \TOOLKIT\SAMPLES\MM\CAPDLL directory) provides functions that drive the display and management of the caption file in a "caption window" in a PM application. The Caption Sample Application (located in the \TOOLKIT\SAMPLES\MM\CAPSAMP subdirectory) demonstrates how an application uses the functions provided by the Caption DLL to take advantage of its services. As with all OS/2 multimedia samples, the source code is provided for all three components. You can use the three components as provided or modify them to meet specific requirements. See Sample Application Programs for more information on the captioning components.

Creating a Caption File

The Caption Creation Utility enables synchronization of the line-by-line display of a text file with the playing of an audio file. You can start this program from either an OS/2 command prompt by typing **CAPTION** (while in the \TOOLKIT\SAMPLES\MM\CAPTION subdirectory), or by selecting the Caption Creation Utility object from the Toolkit folder. You can open an audio file, open a text file to synchronize with it, play the audio file, and select (by clicking a mouse button) the moment in the audio when the current line of text should scroll to display the next line of text. This allows the user to synchronize an audio file with a text file.



```
                        File
```

In order to start the synchronization process, the user selects **Start timing**. The audio file begins playing and **Advance line** is enabled. When you want to scroll to the next line of text, you select **Advance line**. This scrolls the line of text in the text window and displays the next line. When **Advance line** is selected, the Caption Creation Utility issues an MCI_STATUS message with mciSendCommand as shown in the following figure. The device ID passed is obtained when the application opens the audio device. The MCI_STATUS_ITEM flag is set and the *ulItem* field in the MCI_STATUS_PARMS data structure contains MCI_STATUS_POSITION. Upon return, the *ulReturn* field in the MCI_STATUS_PARMS data structure contains the current position of the device in MMTIME units.

```
case ID_NEXTLINE:
   if ( usNextline < usLineCount ) /* (1) Check usNextline.   */
   {
      msp.hwndCallback = (HWND) NULL; /* (2) Get audio
                                    position.           */

      msp.ulItem     = MCI_STATUS_POSITION;
      ulError = mciSendCommand ( mop.usDeviceID,
                                 MCI_STATUS,
                                 MCI_WAIT | MCI_STATUS_ITEM,
                                 (PVOID) &msp,
                                 (USHORT)  UP_STATUS );
```

When the Caption Creation Utility receives the position value, it writes the time value and the line of text to the caption file. A caption file contains a copy of the text file and (before each line of text) the multimedia time unit when that line of text should be displayed in conjunction with the audio file. The file name of the caption file is the same as the file name of the text file, with an extension of ._CC. The caption file can then be used by an application in conjunction with the Caption DLL to caption an application.

Displaying Captions in a Window

The Caption DLL works with an application to display caption files in the requesting application's window. The DLL does all of the work to display the text window and scroll the text in synchronization with the appropriate audio file. Applications can utilize this DLL to display captioned text (in synchronization with an audio file) in their application's window. To perform this function, the Caption DLL utilizes the captioned text files, created by the Caption Creation Utility, and position advise messages that it sets on the specified audio device. As described earlier, the caption files contain an MMTIME unit before every line of text. The Caption DLL requests position advise messages from the Media Device Manager (MDM) every 1500 MMTIME units on the specified device. When it receives MM_MCIPOSITIONCHANGE messages from MDM, it examines the MMTIME units in front of the text lines and displays the correct line of text in the text window of the application. The maximum size of the caption file is 500 lines long, but this can be modified by changing the limit in the Caption DLL source code. This DLL supports the following functions:

ccInitialize — Creates a captioning window and returns the handle of the window to the application.

ccSendCommand — Controls the captioning window, when it has been created, using the following commands: CC_START, CC_STOP, CC_STATUS, and CC_SET.

ccTerminate — Destroys the captioning window and releases any resources allocated for captioning.

See Caption DLL for more information on the captioning functions and data structures.

Caption Sample Application

The Caption Sample Application demonstrates the incorporation of captioning in an application using caption files and the Caption DLL.

As part of its initialization and termination routines, the Caption Sample Application issues ccInitialize and ccTerminate respectively. This notifies the DLL to begin and end captioning.

```
/*
 * Create the caption window and save the handle for further use.
 */
```

```
  hwndCaption = ccInitialize ( (HWND) hwndMainDialogBox );
```

When the ccInitialize function is called, the DLL creates the caption window, but keeps it hidden until the DLL receives a ccSendCommand with a CC_START message.

```
/*
 * Close the captioning system. This will release all the resources
 * that were allocated.
 */
  ccTerminate(hwndCaption);
```

When the user selects **Play**, the Caption Sample Application opens the audio file and obtains a device ID. It then plays the audio file. Finally, it checks the system's captioning flag. If it is set, the Caption Sample Application issues ccSendCommand with a CC_START command. This is all an application must do to implement captioning with OS/2 multimedia. The Caption DLL then starts providing captioning for the application. Three important parameters are sent with this function. First, the device ID or alias is passed. This tells the DLL the correct audio device for which to request position-advise messages. Second, the name of the caption file is passed, and third, the application's window handle is passed. This tells the DLL which caption file to display and the handle of the window to display it in.

```
/*
 * Test the MMPM/2 Captioning Flag.  If it is ON, then the user
 * wants to see captioning.  If it is OFF, the user does not want to
 * see captioning.
 */
mciQuerySysValue ( MSV_CLOSEDCAPTION, &bCCflag );
    .
    .
    .

/*
 * Captioning flag is ON.
 * Fill in the CC_START_PARMS structure and then call ccSendCommand
 * to make the captioning window visible.  The hwndOwner field holds
 * the window handle that we want the Caption DLL to send the
 * position change messages to, when it is done processing them.
 */

csp.pszDeviceName    = (PSZ) "capsamp";     /* Alias name          */
csp.pszCaptionFile   = (PSZ) "CAPSAMP._CC"; /* File name to use    */
csp.hwndOwner        =  hwnd;               /* for position change */

ulReturn = ccSendCommand ( CC_START, MPFROMHWND(hwndCaption), &csp );
                                            /* Start captioning    */
```

If you pause the audio file, change the volume, or move the audio slider position, the Caption Sample Application does not have to do *any* special processing to manage the caption window. The Caption DLL handles this.

If you select **Stop**, the Caption Sample Application issues an MCI_STOP to the audio device, and then it issues a ccSendCommand of CC_STOP to the Caption DLL. This function informs the Caption DLL to stop displaying the caption window in the application and hide the caption window.

```
       case IDC_GPB_STOP:     /* User selected "Stop" push button */
          /*
           * If the audio is not in stopped state, stop the device
           * and hide the text window.
           */
          if (eState != ST_STOPPED)
          {
             StopTheDevice();
             ccSendCommand( CC_STOP, MPFROMHWND(hwndCaption), 0 );
          }
          break;
```

The application issues a ccSendCommand with CC_STATUS to determine the current properties of the caption window. This function initializes the settings dialog box to display it to the user. The following figure shows the status request for the text columns. Requests for the status of the text rows, background color, text color, and window position are handled similarly.

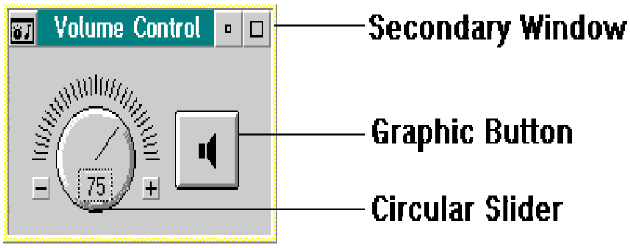
```
   /*
    * Query the current status of the text columns.
    * The CC_STATUS returns the actual value in the ulReturn field.
    */
  ccStatusParms.ulItem = CC_STATUS_TEXT_COLUMNS;
  ccSendCommand( CC_STATUS, MPFROMHWND(hwndCaption), &ccStatusParms );

   /*
    * Get the index value for the ulReturn field.
    */
   if (ccStatusParms.ulReturn == 15)
      ulArrayIndexValue = 0;
   else
   if (ccStatusParms.ulReturn == 35)
      ulArrayIndexValue = 1;
   else
   if (ccStatusParms.ulReturn == 50)
      ulArrayIndexValue = 2;

/*
 * Set the current index value in the spin button.
 */
WinSendDlgItemMsg( hwnd,                 /* Handle to the dialog box  */
                   IDC_TEXT_COLUMNS_SB,  /* ID of the spin button  */
                   SPBM_SETCURRENTVALUE, /* Set current index value */
                   MPFROMLONG(ulArrayIndexValue),/* Current index */
                   NULL );                       /* Ignore        */
```

You can change several properties of the caption window by selecting **Settings** from the **Options** pull-down menu of the Caption Sample Application. When you select **OK** to save the desired properties, the Caption Sample Application issues ccSendCommand with a CC_SET message. The Caption DLL handles changing and displaying the new properties of the caption window. The following figure shows sample code from the Caption Sample Application that sets up the CC_SET_PARMS data structure for the text rows. Changing the settings for the background color, text color, window position, and text columns is handled similarly.

```
   CC_SET_PARMS       ccSetParms;    /* Set parms for CC_SET   */
   ULONG              ulArrayIndexValue=0; /* For spin button return
                                        value             */


    /*
     * Query the text rows spin button. The array index
     * value will be returned in ulArrayIndexValue variable.
     */
    WinSendDlgItemMsg(
       hwnd,
       IDC_TEXT_ROWS_SB,
       SPBM_QUERYVALUE,
       (MPARAM) &ulArrayIndexValue,
       MPFROMLONG(0));

    /*
     * Get the actual value and initialize the CC_SET_PARMS
     * data structure with the appropriate information.
     */
    ccSetParms.ulRows = (ULONG) atoi( textRows[ulArrayIndexValue] );

    /*
     * Issue the CC_SET command with the ccSendCommand and close
     * the dialog box.
     */
    ccSendCommand(CC_SET, MPFROMHWND(hwndCaption), &ccSetParms);
    WinDismissDlg( hwnd, TRUE );

    return( 0 );
```

OS/2 Multimedia Controls

This chapter describes how to create and manage your own customized windows, graphic buttons and secondary windows. OS/2 multimedia applications, such as Volume Control shown in the following figure provide you with examples of multimedia control implementations.



The SW.DLL dynamic link library, located in the \MMOS2\DLL subdirectory contains the functions that support creating and manipulating these controls. (The SW.H file is included with the Toolkit and contains the prototypes, constants, and data structures your application needs.) Refer to the *PM Guide and Reference* for information on creating circular sliders (dials).

Refer to the *CUA Guide to Multimedia User Interface Design* for more information about creating consistent user interfaces for multimedia controls.

Graphic Buttons

A graphic button is a specialized push button that displays text, or graphics, or both. If a graphic button is defined as a two-state graphic button, it can have an up or down appearance. The two-state button remembers its state and thus can be toggled from up to down and down to up. Unlike a standard push button, the graphic button remains in the changed state after the user has clicked on it.

Individual bitmaps can be displayed along with text while the button is in an up, down, highlighted, or unhighlighted state. A graphic button is highlighted when the mouse select button is held down while the pointer is on the graphic button.

Typically, when a two-state button is up, it is selectable, and when it is down, an action is being processed. The bitmap shown in the up state reflects the action to be processed.

If the button has been defined as an animated graphic button, a series of bitmaps can be displayed to produce an animated effect. If the button is a two-state button, animation can be done for a particular button state, or it can be independent of the change of button state. The drawing of animation bitmaps takes precedence over the current state bitmap.

Any graphic button that draws a bitmap while the button is highlighted or unhighlighted is not intended to be animated. Combining an animation button style with either of these two button styles can produce unpredictable results.

Graphic button text can appear as flat on the button surface, or as three-dimensional, with its z-order raised or recessed, relative to the button surface.

These functional capabilities make graphic buttons ideal to use when designing a multimedia device control panel.

Styles

The following table identifies the graphic button styles that can be specified for a graphic button to increase its functionality. A graphic button with no style bits set has an "up" state and the capability to display bitmaps, or text, or both.

Style	Description
GBS_TWOSTATE	Creates a graphic button that has two states: up and down. When the button is in the up state (drawn with its z-order above the owner), this usually indicates it is selectable. When the button is in the down state (drawn with its z-order below the owner), this usually indicates an action is being processed.

```
GBS_AUTOTWOSTATE     Creates a two-state graphic button that
                     automatically toggles its state from up
                     to down or down to up whenever the user
                     clicks on it.  No messages from the
                     owner are required for the button to
                     toggle its state.

GBS_ANIMATION        Creates an animated graphic button that
                     displays a series of bitmaps. Display of
                     the series is handled like a circular
                     list; after the last bitmap in the
                     series is displayed, the first bitmap in
                     the series is displayed, and so forth.

GBS_AUTOANIMATION    Creates an animated graphic button that
                     automatically toggles its animation from
                     off to on or on to off whenever the user
                     clicks on it. Usually, when the button's
                     animation is off, the button is
                     selectable. Conversely, when the
                     animation is on, an action is being
                     processed.

GBS_HILITEBITMAP     Creates a graphic button that displays a
                     different bitmap when the button is in a
                     highlighted paint state.  The
                     highlighted paint state occurs when the
                     user presses the mouse button while the
                     pointer is over the graphic button (or
                     the user holds the spacebar down while
                     the graphic button has the focus).

GBS_DISABLEBITMAP    Creates a graphic button that displays a
                     different bitmap when the highlighted
                     paint state of the button is disabled.

GBS_3D_TEXTRECESSED  Creates a graphic button that has
                     three-dimensional text.  The text
                     z-order is below the face of the button.
                     The main body of the text is black, and
                     its bottom and right edges are white.

GBS_3D_TEXTRAISED    Creates a graphic button that has
                     three-dimensional text.  The text
                     z-order is above the face of the button.
                     The main body of the text is white, and
                     its bottom and right edges are black.
```

Note: Setting both GBS_3D_TEXTRECESSED and GBS_3D_TEXTRAISED is not recommended, because this can cause unpredictable behavior.

Owner Notifications

The owner window of a graphic button is sent a notification code with the WM_CONTROL message whenever the graphic button changes state. The notification code indicates the new state of the button. WM_CONTROL messages are sent with WinSendMsg, which does not return until the application processes the message. These notifications are provided for an application that requires synchronous knowledge of when a graphic button changes state. An example of this type of graphic button is the music scan button shown in the example in section Processing Messages for a CD Player Graphic Button.

At the same time a WM_CONTROL message is sent to the owner, a WM_COMMAND message is also posted to the application message queue with WinPostMsg. The WM_COMMAND informs the application that the graphic button has been selected. If you want to implement a simple graphic button that performs an action, such as a playback operation, you can use code developed for a PM pushbutton, which uses the WM_COMMAND means of notification. The only change you have to make to the code is the window class.

When you create your graphic button, you can take advantage of either of these means of notification.

```
Notification Code        Description
```

GBN_BUTTONUP	The graphic button is changing to an up paint state from either a down or highlighted paint state.
GBN_BUTTONDOWN	The graphic button is changing to a down paint state from either an up or highlighted paint state.
GBN_BUTTONHILITE	The graphic button is changing to a highlighted paint state from either an up or down paint state.

Control Messages

The following table describes the graphic button control messages that can be used by a graphic button window procedure to manipulate a graphic button:

Control Message	Description
GBM_SETGRAPHICDATA	Sets the graphical data (graphic button text, bitmaps) for a graphic button and erases all previous data relating to the state of the button.
GBM_ANIMATE	Sets the animation of an animated graphic button to start or stop at the first bitmap in the series or at a bitmap within the series.
GBM_QUERYANIMATIONACTIVE	Gets the animation state of an animated graphic button.
GBM_SETANIMATIONRATE	Sets, in milliseconds, the period between bitmap updates for an animated graphic button.
GBM_QUERYANIMATIONRATE	Gets the animation rate that is set for an animated graphic button.
GBM_SETSTATE	Sets a two-state graphic button to up or down, or toggles its state.
GBM_QUERYSTATE	Gets the state of a graphic button. Note: For a graphic button that does not have a two-state style, its state is always considered to be "up."
GBM_SETBITMAPINDEX	Sets the bitmap index to use for the various states of the graphic button; up, down, highlighted, not highlighted, beginning of animation series, end of animation series, current state (refers to either the up or down bitmap).
GBM_QUERYBITMAPINDEX	Gets the bitmap index used for a particular button state.

GBM_SETTEXTPOSITION	Sets graphic button text position above or below the bitmap.
GBM_QUERYTEXTPOSITION	Gets graphic button text position relative to the bitmap.

Creating Graphic Buttons

The graphic button PM window class WC_GRAPHICBUTTON is similar to the window class of a push button. This window class must be registered with the function WinRegisterGraphicButton before you can create a graphic button.

A graphic button can be created by a CONTROL statement in a dialog resource. A graphic button also can be created by specifying the WC_GRAPHICBUTTON window class parameter of the WinCreateWindow call.

The graphic button should be initialized when its owner receives a WM_INITDLG message. By doing the initializations at this time, the owner has the capability to change the graphic button's bitmaps, text positioning, state, animation rate, and so on, before the button is displayed on the screen.

The GBTNCDATA data structure shown below is the data structure that is allocated to initialize graphic button control data. This structure is required when sending the GBM_SETGRAPHICDATA message.

Note: If you create a graphic button with WinCreateWindow and initialize the GBTNCDATA structure; set the *usReserved* field to GB_STRUCTURE, rather than GB_RESOURCE, to indicate the structure contains a module handle.

```
typedef struct _GBTNCDATA
{
   USHORT  usReserved;   /* Reserved                       */
   PSZ     pszText;      /* Initial graphic button text    */
   HMODULE hmod;         /* Handle of bitmap resource      */
   USHORT  cBitmaps;     /* Number of button bitmaps       */
   USHORT  aidBitmap[1]; /* Array of bitmap resource IDs   */
} GBTNCDATA;
```

Graphic button data is set or changed by sending the GBM_SETGRAPHICDATA message with WinSendMsg to the graphic button control window procedure. Using this message to change graphic button data erases any data relating to the button state and sets the button state to the default parameters. The default state of a graphic button is "up." If you want to change only the text of the graphic button without affecting the button state data, use WinSetWindowText.

Mnemonics are supported for graphic button text. As with push buttons, a character in the text is designated as the mnemonic for the button by a preceding tilde (˜) character. If the button does not have any text, a null string must be specified.

The number specified for bitmaps associated with the graphic button does not necessarily represent the number of *unique* bitmaps. A graphic button can have duplicate bitmaps associated with it for animation purposes.

It is assumed that all the bitmaps associated with a particular graphic button are of equal size. The size of a graphic button is determined by the dimensions specified in the CONTROL statement of the dialog resource, as well as the size of the bitmap. If the dimensions in the CONTROL statement are too small to contain the bitmap and the text, the button size is made larger to accommodate the width of the bitmap and the depth of the bitmap plus the text. However, if the width of the text exceeds the width determined by the specified dimensions or the actual bitmap (whichever is greater), the text is truncated.

Animated Graphic Button

The following code fragment is a sample definition for creating an animated graphic button with a CONTROL statement in a dialog resource.

```
CONTROL "", IDD_TESTPLAY1, 120, 70, 45, 35,
            WC_GRAPHICBUTTON,
            GBS_AUTOANIMATION | GBS_3D_TEXTRAISED |
            WS_VISIBLE | WS_TABSTOP
            CTLDATA GB_RESOURCE, "PLAY", 16, ID_PLAY1, ID_PLAY 2, ID_PLAY3,
                                               ID_PLAY4,  ID_PLAY5,  ID_PLAY6,
                                               ID_PLAY7,  ID_PLAY8,  ID_PLAY9,
                                               ID_PLAY10, ID_PLAY11, ID_PLAY12,
                                               ID_PLAY13, ID_PLAY14, ID_PLAY15,
                                               ID_PLAY16, 0
```

The control data for the graphic button provides the text "PLAY" (with no mnemonic) for the button face and associates the names of 16 bitmaps with the button. The bitmaps are assigned indexes 1 through 16, according to the order they appear in the control data.

The list of bitmap IDs is preceded by a number and ends with a zero. The number indicates the total of defined bitmaps. The zero indicates the end of the bitmap array. The number of bitmaps that actually are displayed is determined by the bitmap total or the zero-terminated array, whichever is less. If the number of bitmap IDs is greater than the bitmap total, the extraneous bitmap IDs are ignored. A bitmap cannot have an ID of zero.

Because the style of this graphic button includes GBS_AUTOANIMATION, when the user clicks on this button, it automatically toggles the animation on or off without intervention from the owner window of the graphic button.

An animated graphic button can also be created by specifying the WS_GRAPHICBUTTON window class name as a parameter of the WinCreateWindow call. The following code fragment shows an example of setting up the GBTNCDATA structure with the graphic button data and using the WinCreateWindow call.

```
HWND        hwndGB;        /* Graphic button window handle   */
PGBTNCDATA  pgbtn;         /* Pointer to graphic button data */
LONG        lSize;         /* Size of graphic data           */
#define NUMBITMAPS   4     /* Number of bitmaps for button   */

lSize = sizeof(GBTNCDATA) + sizeof(USHORT) * (NUMBITMAPS - 1);
pgbtn = (PGBTNCDATA)malloc(lSize);

if (pgbtn)
 {
  memset(pgbtn, 0, lSize);

  pgbtn pszText     = "Text";
  pgbtn cBitmaps    = NUMBITMAPS;
  pgbtn aidBitmap[0] = BMP0;
  pgbtn aidBitmap[1] = BMP1;
  pgbtn aidBitmap[2] = BMP2;
  pgbtn aidBitmap[3] = BMP3;

  WinRegisterGraphicButton();

  /* Create the graphic button.  hwnd is the window handle
   * of the owning window (for example, client window)
   */

   hwndGB = WinCreateWindow (hwnd,
     WC_GRAPHICBUTTON,
     "",             /* No text here; see pgbtn pszText */
     WS_VISIBLE | WS_TABSTOP | WS_POINTSELECT |
     GBS_AUTOANIMATION | GBS_3D_TEXTRECESSED,
     0,0,80,40,
     hwnd,
     HWND_TOP,
     ID_GB,            /* Graphic button identifier */
     MPFROMP(pgbtn),
     NULL);

  WinSendMsg( hwndGB, GBM_SETGRAPHICDATA, MPFROMP(pgbtn), 0);
}
```

Two-State Graphic Button

The following figure is a sample definition for creating a two-state graphic button in a dialog resource.

```
CONTROL "", IDD_MP_PAUSE, 65, 10, 40, 30,
            WC_GRAPHICBUTTON,
            GBS_TWOSTATE | GBS_3D_TEXTRECESSED |
            WS_VISIBLE | WS_TABSTOP
            CTLDATA GB_RESOURCE, "PAUSE", 3, ID_MP_PAUS0, ID_MP_PAUS1,
                                             ID_MP_PAUS2, 0
```

The control data for the graphic button provides the text "PAUSE" (with no mnemonic) for the button face and associates the names of three bitmaps with the button. The bitmaps are assigned indexes 0, 1, and 2, according to the order they appear in the control data. In this case, the bitmaps are used to indicate the various states of the button: up, down, and highlighted.

Because the button does not have a GBS_AUTOTWOSTATE style, the owner window must send a GBM_SETSTATE message to the graphic button, requesting the button change its state.

A two-state graphic button can also be created by specifying the WC_GRAPHICBUTTON window class name as a parameter of the WinCreateWindow call. The following figure shows an example of setting up the GBTNCDATA structure with the graphic button data and using the WinCreateWindow call.

```
HWND        hwndTSB;    /* Two-state window handle             */
PGBTNCDATA  pgbtn;      /* Graphic button control data         */
LONG        lSize;      /* Size of graphic button control data */
# define NUMBITMAPS  2  /* Number of bitmaps for button */

lSize = sizeof(GBTNCDATA) + sizeof(USHORT) * (NUMBITMAPS - 1);
pgbtn = (PGBTNCDATA)malloc(lSize);

if (pgbtn)
{
  memset(pgbtn, 0, lSize);

  pgbtn pszText    = "Text";
  pgbtn cBitmaps   = NUMBITMAPS;
  pgbtn aidBitmap[0] = BMP0;
  pgbtn aidBitmap[1] = BMP1;

  WinRegisterGraphicButton();

  /* Create the two-state graphic button.  hwnd is the window handle
   * of the owning window (for example, client window)
   */
  hwndTSB = WinCreateWindow (hwnd,
    WC_GRAPHICBUTTON,   /* No text here; see pgbtn pszText */
    "",
    WS_VISIBLE |, WS_TABSTOP | WS_POINTSELECT
    | GBS_AUTOTWOSTATE | GBS_3D_TEXTRECESSED,
    0,0,80,40,
    hwnd,
    HWND_TOP,
    ID_TSB,            /* Graphic button identifier */
    MPFROMP(pgbtn),
    NULL);

  WinSendMsg(hwndTSB, GBM_SETGRAPHICDATA, MPFROMP(pgbtn), 0);

  WinSendMsg(hwndTSB, GBM_SETBITMAPINDEX,
             MPFROMSHORT(GB_DOWN),           /* Which state  */
             MPFROMSHORT(GB_INDEX_LAST));    /* Which bitmap */
}
```

Processing Messages for a CD Player Graphic Button

The following code fragment illustrates a message-handling procedure for a CD player window. This example demonstrates synchronization of a graphic button and control of a CD player.

```
CD_Player_Message_Proc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)
{
    switch (msg)  {
      ...
      case WM_CONTROL:

      /* If the Music Scan GraphicButton sent the notification...   */
          if (SHORT1FROMMP(mp1) == ID_MUSIC_SCAN) {

             switch (SHORT2FROMMP(mp1)) {  /* type of notification */

                case GBN_BUTTONHILITE:  /* button held down */
                   CD_Music_Scan_Start (...);
                   break;

                case GBN_BUTTONUP:      /* button released */
                   CD_Music_Scan_Stop (...);
                   break;
                ...
             }
          ...
          }
          ...
          break;
      ...
   }
}
```

When the music scan button on a physical CD player is held down, it allows the listener to hear the music played at an accelerated rate. In the example shown in the previous figure, the effect of the message procedure is similar. If the user holds the mouse button down and the pointer is over the **Music Scan** graphic button, the CD_Music_Scan_Start function is called. If the mouse button is released, or the pointer is removed from the **Music Scan** graphic button, then the CD_Music_Scan_Stop function is called.

Secondary Windows

A secondary window provides a sizeable and scrollable dialog interface. Secondary window functions are compatible with PM dialog window functions, so it is an easy task to make changes to existing code that uses the PM dialog window functions.

The secondary window uses two frame windows, a standard frame and a dialog frame. The standard frame window offers the standard services-moving, sizing, minimizing, maximizing, and closing-as well as a service for resizing the window to a default size.

The window handle returned by WinLoadSecondaryWindow is the handle to the standard frame. This handle is used when associating a help instance, modifying the title bar or system menu, and doing WinSetWindowPos operations.

The window handle that is used to pass messages to the secondary window procedure pointed to by WinLoadSecondaryWindow is the dialog window handle. This handle is used to access controls on the dialog with the WinWindowFromID function. An application can get a handle to the dialog window by passing the outer frame window handle to WinQuerySecondaryHWND. The dialog window is a child of the standard window's FID_CLIENT window.

When the secondary window is opened, the frame window is set to the default size, which accommodates the dimensions of the dialog window. If the user makes the client window smaller by sizing the frame window, the dialog window is clipped and vertical and horizontal scroll bars appear. Selecting **Default Size** restores the frame window to the optimal size for displaying the dialog. Because the Sizeable Dialog Frame Manager manages the display of the standard window and its scroll bars automatically, the application needs to manage only the dialog window.

The OS/2 multimedia Volume Control application, shown in the beginning of the OS/2 Multimedia Controls section, provides an example of a modeless secondary window implementation.

Compatibility with PM Dialog Window Functions

The design of secondary window functions is very much like the design of dialog window functions. The following table, which lists the secondary window functions, includes the names of PM dialog window equivalent functions.

Secondary Window Function	PM Dialog Window Equivalent	Description
WinLoadSecondaryWindow	WinLoadDlg	Creates a modeless secondary window from a dialog template in a resource.
WinSecondaryWindow	WinDlgBox	Creates a modal secondary window from a dialog template in a resource DLL and returns the result value established by the WinDismissSecondaryWindow call. WinSecondaryWindow combines the functions of WinLoadSecondaryWindow, WinProcessSecondaryWindow, and WinDestroySecondaryWindow.
WinProcessSecondaryWindow	WinProcessDlg	Processes a modal secondary window by dispatching messages while the modal window is displayed.
WinDismissSecondaryWindow	WinDismissDlg	Causes modal WinProcessSecondaryWindow or WinSecondaryWindow calls to return.
WinDestroySecondaryWindow	WinDestroyWindow	Destroys a secondary window.
WinDefSecondaryWindowProc	WinDefDlgProc	Provides the default behavior for a secondary window. A secondary window procedure must reference this function for messages that are not handled explicitly.
WinCreateSecondaryWindow	WinCreateDlg	Creates a Secondary Window from a dialog template in the application's executable file.
WinSecondaryMessageBox	WinMessageBox	Creates a modal window that can be used to display error messages and ask questions.
WinQuerySecondaryHWND	WinQueryWindow	Returns either the handle to the outer frame or inner dialog window of a secondary window, depending on the handle supplied as input.
WinDefaultSize	None	Sizes the dialog window to its recommended, optimal size.
WinInsertDefaultSize	None	Adds the Default Size item to the system menu of a secondary window.

Creating a Secondary Window

WinLoadSecondaryWindow and WinSecondaryWindow create secondary windows from dialog templates in a resource file.

WinSecondaryWindow creates a modal window and supports the processing and destruction of the modal dialog window. WinSecondaryWindow is equivalent to the the following code sequence.

```
hwndSW = WinLoadSecondaryWindow(...);
usResult = WinProcessSecondaryWindow(hwndSW);
WinDestroySecondaryWindow(hwndSW);
```

WinSecondaryWindow and WinProcessSecondaryWindow functions do not return until WinDismissSecondaryWindow is called. If your secondary window procedure handles WM_COMMAND messages, it must call WinDismissSecondaryWindow after calling the WinSecondaryWindow or the WinProcessSecondaryWindow function. Optionally, your window procedure can pass WM_COMMAND messages to WinDefSecondaryWindowProc, which calls WinDismissSecondaryWindow.

WinDismissSecondaryWindow hides the secondary window and returns a result code for a WM_COMMAND message, causing WinSecondaryWindow and WinProcessSecondaryWindow to return. For example, if the user selects the **OK** button, your window procedure passes the DID_OK code with WinDismissSecondaryWindow. Although the secondary window is hidden, it still exists. In the case of WinProcessSecondaryWindow, the window procedure must call WinDestroySecondaryWindow. Before WinSecondaryWindow returns, it destroys the secondary window.

Modeless Secondary Window

WinCreateSecondaryWindow or WinLoadSecondaryWindow can be used to create a modeless secondary window. WinCreateSecondaryWindow creates a secondary window from a dialog template that is stored in the application's executable file. WinLoadSecondaryWindow creates a secondary window from a dialog template that is stored in a dynamic link library.

If the template is a resource in a dynamic link library, the application loads the dynamic link library by calling DosLoadModule, and then loads the dialog by calling WinLoadSecondaryWindow (or WinSecondaryWindow, which calls WinLoadSecondaryWindow). A WM_INITDLG message is sent to the secondary window procedure before WinLoadSecondaryWindow returns.

```
#define INCL_SECONDARYWINDOW             /* Secondary window functions */
#include <sw.h>

PDLGTEMPLATE pdlgt;

DosGetResource (NULL, RT_DIALOG, ID_DIALOG, (PVOID) pdlgt);

WinCreateSecondaryWindow ( HWND_DESKTOP, /*  Parent window     */
                NULL                     /*  Owner window      */
                MyDlgProc                /*  Dialog procedure  */
                pdlgt                    /*  Dialog template   */
                NULL);                   /*  Create parameters */
```

If the template is a resource in the application's executable file, the application loads the resource by calling DosGetResource (as shown in the previous figure) and then uses the template with WinCreateSecondaryWindow to create a secondary window. This method of using a dialog template gives the application the advantage of reviewing and modifying the template before creating the secondary window.

The difference between a modal and a modeless secondary window is the way the windows handle input. For a modal secondary window, WinSecondaryWindow and WinProcessSecondaryWindow handle all user input to the window with an internal message loop and prevent access to other windows in the application. For a modeless secondary window, the application relies on a normal message loop to dispatch messages to the secondary window procedure and does not use WinSecondaryWindow or WinProcessSecondaryWindow.

Secondary Window Message Box

WinSecondaryMessageBox is analagous to WinMessageBox. Both functions create a modal message box that can be used to display error messages and ask questions.

WinSecondaryMessageBox allows more flexibility than WinMessageBox, because you can define the text that appears on the buttons, rather than choosing from a set of standard buttons with predetermined text (**OK**, **Cancel**, and so forth). WinSecondaryMessageBox uses the SMBD and SMBINFO data structures, found in the SW.H file.

The SMBD structure defines the button style, text and ID for each button included in the secondary message box:

```
 typedef struct _SMBD {
    CHAR   achText[MAX_SMBDTEXT + 1]; /* Text of the button, */
                                         for example, "~Cancel". */
    ULONG  idButton;                  /* Button ID returned when user
                                         chooses button. */
    LONG   flStyle;                   /* Button style ORed with
                                         internal styles. */
 } SMBD;
 typedef SMBD * PSMBD;
```

The SMBINFO structure defines the icon used in the message box, specifies the number of buttons in the message box, and points to the array of button definitions.

```
   typedef struct _SMBINFO {
      HPOINTER hIcon;         /* Icon handle                         */
      ULONG    cButtons;      /* Number of buttons                   */
      ULONG    flStyle;       /* Icon style flags (MB_ICONQUESTION) */
      HWND     hwndNotify;    /* Reserved                            */
      PSMBD    psmbd;         /* Array of button definitions         */
   } SMBINFO;
   typedef SMBINFO * PSMBINFO;
```

Adding Default Size to the System Menu

The WinInsertDefaultSize adds the **Default Size** selection to the system menu of the secondary window. This call should be made during the initialization of the secondary window.

When the user selects **Default Size** from the system menu, an SC_DEFAULTSIZE system command is sent to the secondary window procedure, which calls WinDefaultSize to size the window to its optimal default size.

```
#define INCL_SECONDARYWINDOW
#include <sw.h>

HWND hwndFrame

hwndFrame = WinLoadSecondaryWindow (HWND_DESKTOP, /* Parent window  */
                                    HWND_DESKTOP, /* Owner window   */
                                    MyDlgProc,    /* Dialog proc    */
                                    NULL,         /* Module handle  */
                                    ID_DIALOG,    /* Resource ID    */
                                    NULL);        /* Create params  */

WinInsertDefaultSize (hwndFrame, "~Default size"); /* Insert
                                                      menu item     */

WinDefaultSize (hwndFrame);          /* Set window to its
                                        default size                */
```

Multimedia I/O File Services

The multimedia input/output (MMIO) file services are an extension of the base OS/2 file services. Designed to be simple, fast, and flexible, MMIO functions enable an application to access and manipulate multimedia data files in a transparent manner.

Multimedia files contain a variety of media elements such as images, graphics, digital audio and video. These elements can be in different file formats: for example, RIFF, M-Motion, and AVI. Multimedia files can also be stored as memory files, elements of a compound file storage system, or as DOS files. MMIO provides a consistent programming interface so that an application can refer to these files, read and write data to the files, and query the contents of the files, while remaining independent of the underlying file formats or the storage systems that contain the files. In addition, MMIO now enables compressors and decompressors (CODEC procedures) to operate on data objects. See CODEC Procedures for more a more detailed description of CODEC procedures.

MMIO Architecture

Because files services have unique characteristics, the architecture of MMIO is stand-alone and separate from the notion of the media control interface, although some media drivers and file system stream handlers still require file services from MMIO.

When an application calls multimedia I/O functions, the MMIO Manager calls the appropriate I/O procedure (IOProc) if necessary, or processes the function within the MMIO Manager itself. The MMIO Manager uses IOProcs to direct the input and output associated with reading from and writing to different types of storage systems or file formats. IOProcs provide an abstract of the file format, allowing operations such as read, write, and seek to be independent of the specific format in use. The handler is responsible for translating a generic application request into the necessary format-specific operations.

The MMIO architecture provides for CODEC procedures to operate on data objects as required.   A given file format I/O procedure might support none, one, or many CODEC procedures operating on a single format.

The following figure illustrates the architecture of the MMIO subsystem.

```
                                MEDIA           STREAM
             APPLICATIONS      DRIVERS          HANDLER



                             MMIO MANAGER

        File Format IOProcs             File Format IOProcs

                WAVE                        AVC image

              CLI VOC                     M-Motion image

             AVC audio                    1.3 & 2.0 BMP

            RMID & MID                     RDIB & DIB

               AVI                                .
                                                  .
                                                  .


          Codec Procedures                 Codec Procedures

            Ultimotion                         Others

- - - - - - - - - - - - - - - - -   - - - - - - - - - - - - - - -


System        DOS        MEMORY     COMPOUND     OTHERS
Storage       FILE       FILE       FILE
IOProcs
                                                           Ring 3
- - - - - - - - - - - - - - - - - - - - - - - - - -  - - - - - - -
                                                           Ring 0

                            File System
```

Installable I/O Procedures

The MMIO Manager uses I/O procedures to direct the input and output associated with reading and writing to different types of storage systems or file formats. Applications and the MMIO subsystem communicate to IOProcs (DLL files) through the use of MMIO messages. When MMIO receives a request from an application through a function call, messages are created by the MMIO Manager. Next, MMIO sends a predefined message for that operation to the IOProc that supports that particular file format or storage system. These messages are designed for efficient communications to all IOProcs. The IOProcs, however, must be able to handle the messages sent by the MMIO Manager to be processed, or pass them on to a child I/O procedure.

Types of I/O Procedures

Two types of I/O procedures are as follows:

Storage System IOProcs unwrap data objects such as RIFF files, compound RIFF files, or AVC files. These IOProcs are ignorant to the content of the data they contain. A storage system IOProc goes directly to the OS/2 (or native) file system (memory in the case of a MEM file) and does not pass information to any other file format or storage system IOProc. The internal I/O procedures provided for DOS files, memory files, and RIFF compound files are examples of storage system IOProcs because they operate on the storage mechanism rather than the data content itself. See the table in the Internal Storage System Procedures section.

File Format IOProcs manipulate multimedia data at the element level (not to be confused with an element of a RIFF compound file). Each IOProc handles a different element type such as audio, image, or MIDI. A file format IOProc handles the element type it was written for and does not rely on any other file format IOProcs to do any processing. However, a file format IOProc might need to call a storage system IOProc to obtain data within a file containing multiple file elements. For example, the MIDI IOProc calls MMIO functions to access data from other storage system IOProcs supported by MMIO. See the table in the File Format I/O Procedures Provided With OS/2 Multimedia section.

Identifying an I/O Procedure

Each file format is represented by a unique identifier called a four-character code (FOURCC). A FOURCC is a 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters (padded on the right with blank characters).

Each IOProc supports a specific file format. The file format and IOProc are represented by a specific FOURCC code. This permits the FOURCC to be used as an ID value, rather than the string-name of the file format or a file name extension. Their use is supported by a set of functions to pack or unpack FOURCC values from or to their component characters. Examples of FOURCC values are: `WAVE` for RIFF waveform audio files, `RMID` for RIFF MIDI files, and `AVCA` for AVC audio files.

Formats that support multiple media types require a different FOURCC for each variation. This appears as a different IOProc for each media type. For example, an Audio Visual Connection (AVC) program might have an IOProc to process image, an IOProc to process audio, and an IOProc to process MIDI. You can, however, include more than one IOProc in a dynamic-link library (DLL) file by providing different entry points in the DLL file.

The data type for a four-character code is FOURCC. The mmioFOURCC function converts four characters to a four-character code as shown:

```
FOURCC   fccIOProc;

fccIOProc = mmioFOURCC( 'A', 'V', 'C', 'A' ) ;
```

The mmioFOURCC function is called by the application and passed in subsequent MMIO calls to route the file to the correct IOProc.

Note: Another way to create a four-character code is to use the mmioStringToFOURCC function, which converts a null-terminated string to a four-character code. The second parameter in mmioStringToFOURCC specifies options for converting the string to a four-character code. If you specify the MMIO_TOUPPER flag, mmioStringToFOURCC converts all alphabetic characters in the string to uppercase.

Internal Storage System Procedures

The following table lists the MMIO services provided by internal I/O procedures:

IOProc	Description
DOS	Handles standard OS/2 disk files.
MEM (memory)	Manages memory files without accessing the file system. A memory file is a block of memory that is perceived as a file to an application. This unifies the interface for applications that access both files and memory.
CF (compound file)	Supports the RIFF compound file format. The CF IOProc operates on a compound file. The MMIO Manager provides services to find, query, and access file elements in a compound file. It also supports the function of file compaction.

File Format I/O Procedures Provided With OS/2 Multimedia

The MMIO Manager calls a file format I/O procedure to handle I/O to files of a certain media type and format: for example, AVC or M-Motion files. File format I/O procedures are available with the installation of OS/2 multimedia. These I/O procedures are enabled for data and file format translation. They provide conversion support for the Multimedia Data Converter program. By installing file format I/O procedures, existing applications no longer need to store multiple copies of the same media file for running on various platforms using different file formats.

The information in the following table serves as a guide to application developers who would like to access the functions that deal with particular file formats. OS/2 multimedia provides the following file format IOProcs, which can be used to access non-RIFF data and perform multimedia data conversions. Each I/O procedure can read or write format-specific data or standardized data.

IOProc	FOURCC	Description	Common Extension
AIFF	AIFF	Supports AIFF waveform digital audio files.	.AIF
AVC Audio	AVCA	Supports IBM Audio Visual Connection (AVC) digital audio files of type ADPCM and native ACPA formats.	._AU, ._AD
AVC Image	AVCI	Supports IBM Audio Visual Connection (AVC) digital image files.	._IM, .!IM, ._ID
AVI Movie	AVI	Supports audio/video interleaved (AVI) movie files.	.AVI
CLI VOC	VOC	Supports Creative Technology Voice files.	.VOC

DIB	WI30	Supports device independent bitmap image files.	.DIB
FLC/FLI Animation	FLIC	Supports multi-track read requests from AutoDesk animation files.	.FLC, .FLI
GIF	GIFC	Supports compressed GIF image files.	.GIF
JPEG Still Image	JPEG	Supports translated read and write access to JPEG still image files.	.JPG
OS/2 1.3 Bitmap	OS13	Supports OS/2 1.3 and Windows 3.0 uncompressed bitmap image files.	.BMP
OS/2 2.0 Bitmap	OS20	Supports OS/2 2.0 and Windows 3.0 1, 4, 8-bit palettized and 24-bit RGB bitmap image files.	.BMP
MIDI	MIDI, RMID	Supports MIDI files (format 0 and format 1 data), in RIFF or non-RIFF format.	.MID
M-Motion Still	MMOT	Supports translated and untranslated access to IBM M-Motion/M-Control YUV video still image files of type packed 12-bit YUV data.	.VID
MPEG-1 Movie	MPEG	Supports multi-track read requests from an MPEG-1 movie file.	.MPG
PCX	PCXC	Supports compressed PCX image files.	.PCX
PhotoCD	PCD	Supports translated read-only access to PhotoCD image files.	.PCD
RIFF DIB	RDIB	Supports RIFF device independent bitmap image files.	.RDI
RIFF Waveform	WAVE	Supports RIFF waveform digital audio files, including PCM, IBM ADPCM, IBM Mu-Law, and A-Law.	.WAV
TARGA	TGAU, TGAC	Supports uncompressed and compressed TARGA image files.	.TGA
TIFF	TFIU, TFIC, TFMU, TFMC, TFFC	Supports compressed and uncompressed Intel or Motorola TIFF image files and compressed TIFF FAX image files.	.TIF
UNIX SND	SND	Supports UNIX (NeXT/Sun) SND digital audio files.	.SND

Installing an I/O Procedure

Certain factors must be considered when installing an IOProc; for example, the number of processes that use the IOProc might help you decide which method you use to install an IOProc. Depending on the requirements of your application, you can choose to install an IOProc on a temporary, semipermanent, or permanent basis.

The following MMIO functions allow you to install an IOProc:

Function	Description
mmioOpen	Temporarily installs an I/O procedure.

```
 mmioInstallIOProc              Adds, replaces, finds, or
                                removes an entry in the MMIO
                                IOProc table.

 mmioIniFileHandler             Adds, replaces, finds, or
                                removes an entry in the
                                initialization file
                                (MMPMMMIO.INI).
```

Temporary Installation Using mmioOpen

You can temporarily install an I/O procedure using mmioOpen. In this case, the IOProc is only used when a file opened by the MMIO Manager does not install the IOProc in the I/O procedure table.

To specify an I/O procedure when you open a file using mmioOpen, use the *pmmioinfo* parameter to reference an MMIOINFO structure as follows:

1. Load the DLL using the DosLoadModule and DosQueryProcAddr functions to install and obtain the procedure address of the IOProc as shown in the following code fragment.

```
strcpy( acMMIOProcName,
        acStringBuffer[ IDS_MMIO_INSTALLPROC_NAME - 1 ] );

ldosAPIRc =
  DosLoadModule(
   &acFailureNameBuffer[ 0 ],    /* Object name if failure occurs.  */
   FILE_NAME_SIZE,               /* Size of the name buffer.        */
   acStringBuffer[ IDS_MMIO_INSTALLPROC_DLL_NAME - 1 ], /* DLL Name.*/
   &hmModHandle );               /* Handle to the module.           */
     .
     .
     .

ldosAPIRc =
  DosQueryProcAddr(
   hmModHandle,                /* Handle to the DLL module.         */
   (ULONG) NULL,               /* NULL gives the entry point.       */
   acMMIOProcName,             /* Name of the Installable procedure.*/
   (PFN *) &pmmioprocIoProc ); /* Pointer to the Installable proc.  */
```

2. In the MMIOINFO structure passed to mmioOpen, store the procedure address of the IOProc type in the *pIOProc* field. Set the *fccIOProc* field to NULL.

3. Call mmioOpen to use the IOProc with a file, passing the MMIOINFO structure in as the second parameter. This sends a MMIOM_OPEN message to the temporary IOProc. (Note that this does not make the IOProc available for use with other files.)

4. Set all other fields to 0 (unless you are opening a memory file, or directly reading or writing to the file I/O buffer).

This strategy allows a file format IOProc to replace a default IOProc (such as the DOS IOProc), simply by using the address of a replacement custom routine.

Semipermanent Installation Using mmioInstallIOProc

You can install an IOProc during run-time from your application. This method is semipermanent because the IOProc can only be called while the process is active. When the process terminates, it is removed from the IOProc table. Once the DLL is removed from memory, the next loading of the MMIO DLL does not load this IOProc. Because this IOProc is installed by specifically calling mmioInstallIOProc, as shown in the following figure, the IOProc is available for any files opened within that process.

```
PMMIOPROC  pmmioprocSpecialIOProc; /* Pointer once IOProc
                                      is installed.       */


 pmmioprocSpecialIOProc =
   mmioInstallIOProc(
    fccIOProc,          /* The identifier (FOURCC) of the procedure.*/
    pmmioprocIoProc,    /* Pointer to the installable procedure.    */
    MMIO_INSTALLPROC ); /* Flag to install the procedure.           */
```

The mmioInstallIOProc function maintains a separate list of installed I/O procedures for each OS/2 application that uses MMIO. This allows different applications to use the same I/O procedure identifier for different I/O procedures without causing conflict. When you install an I/O procedure using mmioInstallIOProc, the procedure remains installed until you remove it. The mmioInstallIOProc function does not prevent an application from installing two different I/O procedures with the same identifier, or installing an I/O procedure with the same identifier as a internal I/O procedure (DOS, MEM, or CF). When mmioInstallIOProc is called with the MMIO_REMOVEPROC flag set, as shown in the following code fragment, the most recently installed procedure is the first one to be removed.

```
PMMIOPROC  pmmioprocSpecialIOProc; /* Pointer once IOProc
                                      is installed.               */

 pmmioprocSpecialIOProc =
   mmioInstallIOProc(
     fccIOProc,               /* The identifier of the procedure. */
     pmmioprocIoProc,         /* Pointer to the Installable proc. */
     MMIO_REMOVEPROC );       /* Flag to deinstall the proc.      */
```

Permanent Installation Using mmioIniFileHandler

You can permanently install an IOProc in your system by identifying the IOProc in the initialization file (MMPMMMIO.INI) when you start your system. This method allows the IOProc to be used by any process because it is installed in the IOProc table for every process (like the internal I/O procedures). Therefore, a call to mmioInstallIOProc is not necessary every time the IOProc is needed.

The advantage of installing I/O procedures in the MMPMMMIO.INI file is to achieve application transparency; I/O procedures become built-in as soon as you restart your system. Note that the IOProc must be contained in a DLL file, although more than one IOProc can be contained in the DLL if necessary.

To permanently install an IOProc, an IOProc entry is added to the MMPMMMIO.INI file. This is accomplished by either writing an INI change control file or writing an application using the mmioIniFileHandler function with the MMIO_INSTALLPROC specified. The IOProc is installed in the IOProc table ahead of the MMIO default IOProcs (DOS, MEM, and CF).

The following code fragment is an example of how an application uses the mmioIniFileHandler function to permanently install the OS/2 1.3 PM bitmap image IOProc.

```
#define FOURCC_OS13   mmioFOURCC( 'O', 'S', '1', '3' )

#pragma linkage( mmioIniFileHandler, system )

void main ()
{
 ULONG   ul;
 MMINIFILEINFO  mminifileinfo;
 mminifileinfo.fccIOProc = FOURCC_OS13;
 strcpy (mminifileinfo.szDLLName, "OS13PROC");
 strcpy (mminifileinfo.szProcName, "OS13BITMAPIOPROC");
 mminifileinfo.ulExtendLen = 16L;
 mminifileinfo.ulFlags = 0L;
 mminifileinfo.ulMediaType = MMIO_MEDIA_IMAGE;
 mminifileinfo.ulIOProcType = MMIO_IOPROC_FILEFORMAT;
 strcpy (mmioinifileinfo.szDefExt, "");

 printf ("Installing OS/2 PM Bitmap (V1.3) IOProc\n");

 rc = mmioIniFileHandler (&mminifileinfo, MMIO_INSTALLPROC);
 switch (rc)
 {
 case MMIO_SUCCESS:
  printf ("Installing Complete\n");
  break;
 case MMIOERR_INVALID_PARAMETER:
  printf ("Error in this install program\n");
  break;
 case MMIOERR_INTERNAL_SYSTEM:
  printf ("OS/2 MPM System Error\n");
  break;
 case MMIOERR_NO_CORE:
  printf ("Memory unavailable for this IOProc\n");
  break;
 case MMIOERR_INI_OPEN:
  printf ("Unable to access the OS/2 MMPMMMIO.INI file\n");
  break;
 case MMIOERR_INVALID_FILENAME:
  printf ("Cannot find the file : OS13PROC.DLL\n");
  break;
 default:
  printf ("Unknown error attempting to install OS/2 Bitmap V(1.3)\n");
  break;
 }
}
```

CODEC Procedures

CODEC procedures are similar to I/O procedures (IOProcs). They are dynamic-link library (DLL) routines that operate on data within a file or buffer. Based on the data content, a particular CODEC procedure is loaded typically by a file format IOProc to either compress or decompress the data

OS/2 multimedia currently supports image and digital video CODECs. The following table describes the video CODEC procedures provided with OS/2 multimedia.

CODEC	Format	FOURCC	Description
ULDC	Ultimotion	ULTI	Decompressor
INDEO	Indeo 2.1	rt21/RT21	Compressor/Decompressor
INDDEC32	Indeo 3.1/3.2	iv31/IV31/iv32/IV32	Decompressor
ULCORT	Ultimotion	ULTI	Real-time compressor
ULCOASYM	Ultimotion	ULTI	Frame-step compressor
INDRTR31	Indeo 3.1	IV31	Real-time compressor
INDFSR31	Indeo 3.1	IV31	Frame-step compressor
MONDO	Uncompressed	DIB	Supports SW monitor/RGB16-YUV411 playback
AUTOPROC	FLI/FLC	flic	Supports AutoDesk Animator playback
MPGDC	MPEG	MPEG	MPEG hardware interface CODEC

OS/2 multimedia also provides a set of audio CODECs; however, there is no public interface to install new audio CODECs or to interface to audio CODECs directly.

The following table describes the audio CODEC procedures used internally within OS/2 multimedia for playing files that contain compressed audio.

CODEC	Format	Format Tag	Comp.	Decomp.	Description
IMAADPCM	MS ADPCM	2	X	X	Compression is not real-time
IMAADPCM	IMA ADPCM	17	X	X	Real-time compression

CODEC Functions, Messages, and Data Structures

There are enough differences between CODEC procedures and I/O procedures that several additional functions and messages are needed for CODEC processing. Both applications and I/O procedures can call these CODEC functions and messages.

The MMIO subsystem supports the CODEC functions listed in the following table.

Function	Description
mmioIniFileCODEC	Adds, replaces, removes, or finds a CODEC entry in the MMPMMMIO.INI file.
mmioQueryCODECName	Queries the ID string of a CODEC procedure.
mmioQueryCODECNameLength	Queries the length of a CODEC procedure ID string.
mmioLoadCODECProc	Loads the CODEC procedure and returns the entry point.
mmioSet	Sets or queries extended file information.

The messages listed in the following table are supported by I/O procedures.

Message	Description
MMIOM_COMPRESS	Sent to an IOProc to compress data.
MMIOM_DECOMPRESS	Sent to an IOProc to decompress data.
MMIOM_MULTITRACKREAD	Sent to an IOProc to read multi-track data from a file.
MMIOM_MULTITRACKWRITE	Sent to an IOProc to writes multi-track data to a file.
MMIOM_SET	Sets and queries CODEC attributes of the CODEC procedure.

The messages listed in the following table are supported by CODEC procedures.

Message	Description
MMIOM_CODEC_OPEN	Opens a CODEC instance.
MMIOM_CODEC_CLOSE	Closes a CODEC instance.
MMIOM_CODEC_QUERYNAME	Queries the CODEC procedure ID string.
MMIOM_CODEC_QUERYNAMELENGTH	Queries the length of the CODEC procedure ID string.
MMIOM_CODEC_COMPRESS	Compresses data.
MMIOM_CODEC_DECOMPRESS	Decompresses data.

The data structures listed in the following table are supported by I/O procedures and CODEC procedures. The CODEC.H and MMIOOS2.H header files define these data structures. Include the CODEC.H and MMIOOS2.H header files and define INCL_MMIO_CODEC in any source that uses CODEC functions.

Data Structure	Description
MMEXTENDINFO	Used by mmioSet and MMIOM_SET.
MMMULTITRACKREAD	Used by MMIOM_MULTITRACKREAD.
MMMULTITRACKWRITE	Used by MMIOM_MULTITRACKWRITE.
MMMOVIEHEADER	Standard presentation movie header.

MMVIDEOHEADER	Standard presentation video header.
MMVIDEOOPEN	Used by MMIOM_OPEN.
MMCOMPRESS	Used by MMIOM_CODEC_COMPRESS and MMIOM_COMPRESS.
MMDECOMPRESS	Used by MMIOM_CODEC_DECOMPRESS and MMIOM_DECOMPRESS.
CODECVIDEOHEADER	Used by MMIOM_CODEC_OPEN.
CODECOPEN	Used by MMIOM_CODEC_OPEN.
CODECINIFILEINFO	Used by mmioIniFileCODEC.

Installing a CODEC Procedure

Unlike I/O procedures that have several methods of installation, CODEC procedures are installed by registering them in the MMPMMMIO.INI file using the mmioIniFileCODEC function. Once registered in the MMPMMMIO.INI file, queries can be made on CODEC properties associated with the file format of the opened file. A particular CODEC procedure can be associated with different file formats. A separate entry is required for each different file format a CODEC procedure supports. Allowing CODEC procedures to be queried and dynamically loaded based on the data content alleviates the need to statically load CODEC procedures with the file format IOProc, or hard code DLL names with the IOProc. The following code fragment shows how to install a CODEC procedure using mmioIniFileCODEC.

```
CODECINIFILEINFO codecIniFileInfo;
ULONG ulFlags = 0L;
ULONG rc;
      .
      .
      .
memset( &codecIniFileInfo, '\0', sizeof(CODECINIFILEINFO) );
codecIniFileInfo.ulStructLen = sizeof(CODECINIFILEINFO);
codecIniFileInfo.fcc = FOURCC_MYPROC;
codecIniFileInfo.ulCompressType = COMPRESSTYPE_MYPROC;
codecIniFileInfo.ulCompressSubType = COMPRESSSUBTYPE_MYPROC;
codecIniFileInfo.ulMediaType = MEDIATYPE_MYPROC;
codecIniFileInfo.ulCapsFlags = CODEC_DECOMPRESS;
codecIniFileInfo.ulFlags = 0;
codecIniFileInfo.szHWID = HWID_MYPROC;
codecIniFileInfo.ulMaxSrcBufLen = MAXBUFLEN_MYPROC;
codecIniFileInfo.ulSyncMethod = 1;
codecIniFileInfo.ulReserved1 = 0;
codecIniFileInfo.ulXalignment = XALIGNMENT_MYPROC;
codecIniFileInfo.ulYalignment = YALIGNMENT_MYPROC;
strncpy( codecIniFileInfo.szDLLName, "MYPROC.DLL", DLLNAME_SIZE );
strncpy( codecIniFileInfo.szProcName, "MyCODECProc", PROCNAME_SIZE );
ulFlags = MMIO_INSTALLPROC | MMIO_MATCHCOMPRESSTYPE |
          MMIO_MATCHCOMPRESSSUBTYPE |
          MMIO_MATCHCAPSFLAGS | MMIO_MATCHHWID;

rc = mmioIniFileCODEC( &codecIniFileInfo,
                        ulFlags);
if (rc)
  /* error */
else
      .
      .
      .
```

MMIO Data Structures

The MMIOOS2.H header file, supplied with the Toolkit, defines data structures used in passing information between an application and MMIO file services. Include the MMIOOS2.H file in any source that uses multimedia file I/O functions. Also include the main header file, OS2.H, which contains a declaration statement for the MMIOOS2.H file. The MMIOOS2.H file defines MMIO data structures listed in the following table.

Note: The MMMIDIHEADER data structure is defined in the MIDIOS2.H file.

Data Structure	Description
MMAUDIOHEADER	Contains descriptive information about a digital audio element (such as the sample rate, bits per sample, and so forth).
MMCFINFO	Contains information about a RIFF compound file.
MMCKINFO	Contains information about a chunk in a RIFF file.
MMCTOCENTRY	Contains information about an entry in the compound file table of contents (CTOC) of a RIFF compound file.
MMFORMATINFO	Contains information about the format of a file or file element.
MMIMAGEHEADER	Contains descriptive information about an image (such as the size, color type and extent, palette information, and so forth).
MMINIFILEINFO	Contains information about the MMPMMMIO.INI file.
MMIOINFO	Describes the current state of an open file.
MMMIDIHEADER	Contains descriptive information about a MIDI file.

Note: Initialize all fields in MMIO data structures that are not used in a given function to NULL. In addition, set unused flags and bits in flags that are not used to 0.

Basic I/O Functions

The following MMIO functions enable applications to manage basic multimedia file I/O operations. These functions, together with media independent operations, provide applications independence from a specific media type. Basic multimedia file I/O services are shown in the following table.

Function	Description
mmioOpen	Opens or creates a file for reading, writing, or both.
mmioRead	Reads a specified number of bytes from an open file.
mmioWrite	Writes a specified number of bytes to an open file.
mmioSeek	Changes the current position for reading, writing, or both in an open file.

mmioClose	Closes an open file.

Opening or Creating a File

To perform I/O operations on a new or existing file, an application must first establish a connection by calling the mmioOpen function. This returns a file handle, which identifies the open file when calling other file I/O functions. Specify the *pmmioinfo* parameter to identify a pointer to a caller-provided MMIOINFO block containing extra parameters used by mmioOpen. *pmmioinfo* may be NULL if the default values of the fields of *pmmioinfo* are sufficient. By specifying an MMIOINFO structure, you can open a memory file, specify a file format I/O procedure, supply a buffer, or specify data transfers in a standard presentation format. For basic file I/O services, set *pmmioinfo* to NULL. For example, the following code sample shows how to open a data file without a buffer.

```
hmmio = mmioOpen("filename.ext", NULL, 0);
```

Basic options for mmioOpen are shown in the following table.

Flag	Description
MMIO_ALLOCBUF	System allocates an internal I/O buffer.
MMIO_APPEND	Allows appending to the end of a file.
MMIO_BUFSHARED	Uses shared memory if MMIO allocates an I/O buffer.
MMIO_CREATE	Creates a new file.
MMIO_DELETE	Deletes the file if it already exists.
MMIO_NOIDENTIFY	Does not attempt to identify the file type.
MMIO_READ	Opens a file for reading only (default).
MMIO_READWRITE	Opens a file for both reading and writing.
MMIO_VERTBAR	Uses the vertical bar character (|) as a file separator character rather than the plus sign (+).
MMIO_WRITE	Opens a file for writing only.

The following example creates a new file named NEW.TXT and opens it for writing:

```
hmmio = mmioOpen("NEW.TXT", NULL, MMIO_CREATE | MMIO_WRITE);
```

You can also use mmioOpen to specify sharing options for data files. These options enable DOS files to be opened and accessed by more than one process, allowing an application to take advantage of the multi-processing feature of the OS/2 operating system.

Sharing options for data files are shown in the following table.

Flag	Description
MMIO_DENYNONE	Opens a file without denying other processes

```
                read or write access to the file.

 MMIO_DENYREAD   Opens a file and denies other processes read
                 access to the file.

 MMIO_DENYWRITE  Opens a file and denies other processes write
                 access to the file.

 MMIO_EXCLUSIVE  Opens a file with exclusive mode, denying
                 other processes both read and write access to
                 the file.
```

Reading from and Writing to a File

Use the mmioRead and mmioWrite functions to read and write to files opened by mmioOpen. Use the *pch* parameter to specify the buffer to read to (mmioRead) or the buffer to write from (mmioWrite). Use the *cch* parameter to specify the number of bytes to read to or write from *pch* to the file. (The read and write operations are not limited to 64KB.)

The following example shows how to read 20 bytes from a file.

```
mmioRead(hmmio, achBuffer, 20)
```

The following example shows how to write 20 bytes to a file.

```
mmioWrite(hmmio, achBuffer, 20)
```

Note: Where translation (MMIO_TRANSLATEDATA) is enabled, the translation takes place during the read and write operation.

Changing the Position in a File

Use the mmioSeek function to change the current position of the file pointer in an open file. This is the place where the next read or write operation is going to take place. The *lOffset* parameter specifies an offset to move the file position to. The *lOrigin* parameter specifies how the offset is interpreted.

The following example illustrates how to seek to the beginning of an open file.

```
mmioSeek(hFile, 0L, SEEK_SET);
```

To seek to the current file position:

```
mmioSeek(hFile, 0L, SEEK_CUR);
```

To seek to the end of a file:

```
mmioSeek(hFile, 0L, SEEK_END);
```

To seek to a position 20 bytes from the end of an open file:

```
mmioSeek(hFile, -20L, SEEK_END);
```

The return value is the new file position (specified in bytes) from the beginning of the file. If an error occurs, the return code is MMIO_ERROR. Use caution when seeking past the end of the file. Instead of returning MMIO_ERROR, mmioSeek returns the offset of the new file position.

Note: The mmioSeek function supports seeking in translation mode within the limits and capabilities of the specific file format IOProc that handles that format.

Retrieving an Error

Use the mmioGetLastError function to return the last error condition stored in the system for this process. This function assists you in obtaining additional information about the failing condition for error routine analysis.

Note: This function only works if you have an MMIO handle.

Buffered File I/O

Using buffers improves the performance of multimedia applications that perform numerous file I/O operations when accessing media devices. With buffered file I/O, the system maintains a block of memory that the operating system uses to store data as it is read from (or written to) the disk. If data is already in the buffer, the operating system can transfer the record to the application's area without reading the sector from the disk. This improves performance by reducing the number of times the application accesses the disk for read or write operations. The application only accesses the media device when the buffer must be filled from or written to the disk.

Many factors must be considered when deciding whether or not to use buffered file I/O; for example, the number and size of read and write operations. Although it is difficult to identify criteria for using buffered I/O, there are some general guidelines you can follow. It is a good idea to use buffered I/O for applications that perform many I/O operations, less than 4KB each. However, if you are performing I/O operations where buffer sizes may be larger than 4KB (as in the case of streaming), it might be best to use unbuffered I/O. If an application such as the Sync/Stream Manager (SSM) provides its own internal buffers, a secondary set of buffers during run-time may hinder, rather than improve, performance. Experiment to optimize file I/O for your application's requirements.

Opening a File Using Buffered File I/O

When a file is opened for buffered I/O, the buffer is essentially transparent to the application. You can read, write, and seek in the same way as unbuffered I/O. To open a file using buffered file I/O, you can either provide an I/O buffer in an application, or allow the system to allocate an internal buffer. To provide a user-supplied I/O buffer, you can either use mmioOpen and have the *pmmioinfo* field point to the buffer, or use mmioSetBuffer (see the table of buffered I/O functions in the next subsection.) To provide a system-supplied buffer, specify the MMIO_ALLOCBUF option of the mmioOpen function. Unless you have a performance-sensitive application that directly accesses an I/O buffer or opens a memory file, it is a good idea to use the MMIO Manager to allocate the buffer. For example, the following code sample opens a DOS file and directs mmioOpen to allocate a standard-sized buffer.

```
hmmio = mmioOpen("EXAMPLE.DIB", NULL, MMIO_ALLOCBUF);
```

Managing Buffered I/O

The MMIO functions shown in the following table allow you to manage an I/O buffer.

Function	Description
mmioFlush	Forces the contents of an I/O buffer to be written to disk.
mmioSetBuffer	Enables or disables buffered I/O, and changes the buffer or buffer size for an open file.

Emptying the Contents of an I/O Buffer

Emptying the contents of an I/O buffer means that the contents of the buffer are written to disk. You can empty the contents of a buffer by calling mmioFlush or mmioClose. The buffer is automatically emptied when you close a file by calling mmioClose. If you do not close a file immediately after writing to it, empty the contents of the buffer to make sure the information is written to disk. You can also use the MMIO_EMPTYBUF flag of mmioFlush to clear the I/O buffer without deallocating the buffer.

Note: The mmioFlush function may fail if there is insufficient disk space to write the buffer, even if the preceding mmioWrite functions succeeded.

Setting or Changing an I/O Buffer

Use the mmioSetBuffer function to enable or disable I/O buffering for reading to or writing from files. You can also change the size of the internal I/O buffer (8KB default) or supply your own buffer for use as a memory file.

The mmioSetBuffer function requires a *pchBuffer* parameter, which identifies the pointer to a user-supplied buffer for buffered I/O. If you want mmioSetBuffer to allocate the buffer, or if you want to disable any predefined I/O buffers, set *pchBuffer* to NULL. A second parameter, *cchBuffer*, specifies the size of the caller-supplied buffer. If you set *pchBuffer* to NULL, *cchBuffer* is the size of the buffer that the you want mmioSetBuffer to allocate. To disable buffering, set *pchBuffer* and *cchBuffer* to NULL.

The following example illustrates how to open an unbuffered file named TESTING and then allocate an internal 16KB buffer.

```
HMMIO hFile;
.
.
.
if ((hFile = mmioOpen("TESTING", NULL, MMIO_READ)) !=NULL) {
   /* File opened successfully; request an I/O buffer */
   if (mmioSetBuffer(hFile, NULL, 16384L, 0))
       /* Buffer cannot be allocated */
   else
       /* Buffer allocated successfully */
}
else
     /* File cannot be opened */
```

The following example illustrates how to open a buffered file named TESTING and then disable buffered I/O.

```
HMMIO hmmio;
.
.
```

```
.
if ((hmmio = mmioOpen("TESTING", NULL, MMIO_ALLOCBUF)) !=NULL) {
   /* File opened successfully; disable buffered I/O */
   if (mmioSetBuffer(hFile, NULL, NULL, 0))
       /* Cannot disable buffered I/O  */
   else
       /* Buffered I/O disabled successfully */
}
else
       /* File cannot be opened */
```

Directly Accessing a File I/O Buffer

Applications that are performance-sensitive can optimize file I/O performance by directly accessing the file I/O buffer. Exercise care if you choose to do this-by accessing the file I/O buffer directly, you bypass some of the safeguards and error checking provided by the MMIO Manager.

The MMIO functions shown in the following table allow you to support direct I/O buffer access on a file opened for buffered I/O.

Function	Description
mmioGetInfo	Retrieves information on the file I/O buffer of a file opened for buffered I/O.
mmioAdvance	Fills and empties the contents of an I/O buffer of a file set up for direct I/O buffer access.
mmioSetInfo	Changes information on the file I/O buffer of a file opened for buffered I/O.

Note: After you call mmioGetInfo, do not call any MMIO functions other than mmioAdvance. You can begin calling MMIO functions again after you call the mmioSetInfo function.

The MMIO Manager uses the MMIOINFO data structure to maintain state information on an open file. The MMIOINFO data structure is defined in the MMIOOS2.H header file as as shown.

```
typedef struct _MMIOINFO {       /* mmioinfo                    */
   ULONG       ulFlags;          /* Open flags                  */
   FOURCC      fccIOProc;        /* FOURCC of the IOProc to use */
   PMMIOPROC   pIOProc;          /* Function Pointer to IOProc to use */
   ULONG       ulErrorRet;       /* Extended Error return code  */
   LONG        cchBuffer;        /* I/O buff size (if used), Fsize if MEM */
   PCHAR       pchBuffer;        /* Start of I/O buff           */
   PCHAR       pchNext;          /* Next char to read or write in buff */
   PCHAR       pchEndRead;       /* Last char in buff can be read + 1  */
   PCHAR       pchEndWrite;      /* Last char in buff can be written + 1 */
   LONG        lBufOffset;       /* Offset in buff to pchNext */
   LONG        lDiskOffset;      /* Disk offset in file       */
   ULONG       aulInfo[4];       /* IOProc specific fields    */
   LONG        lLogicalFilePos;  /* Actual file position, buffered or not */
   ULONG       ulTranslate;      /* Translation field         */
   FOURCC      fccChildIOProc;   /* FOURCC of Child IOProc    */
   PVOID       pExtraInfoStruct; /* Pointer to a structure of related data */
   HMMIO       hmmio;            /* Handle to media element   */
   } MMIOINFO;
```

Getting Buffer Information

Use mmioGetInfo to obtain information about a file I/O buffer, such as the buffer size and address. The mmioGetInfo function also sets up a file for direct I/O buffer manipulation.

The mmioGetInfo function identifies a pointer to an MMIOINFO structure that mmioGetInfo fills with information about the file I/O buffer. The return value is 0 if the operation is successful; otherwise, the return value specifies an error code.

Reading from and Writing to the Buffer

The following table shows three fields in the MMIOINFO structure used for reading from and writing to the file I/O buffer.

Field	Description
pchNext	Points to the next location in the buffer to read or write. You must increment *pchNext* as you read and write the buffer.
pchEndRead	Identifies the location containing the last valid character you can read from the buffer. This is the memory location following the last valid data in the buffer.
pchEndWrite	Identifies the last location in the buffer you can write to. This is the memory location following the end of buffer.

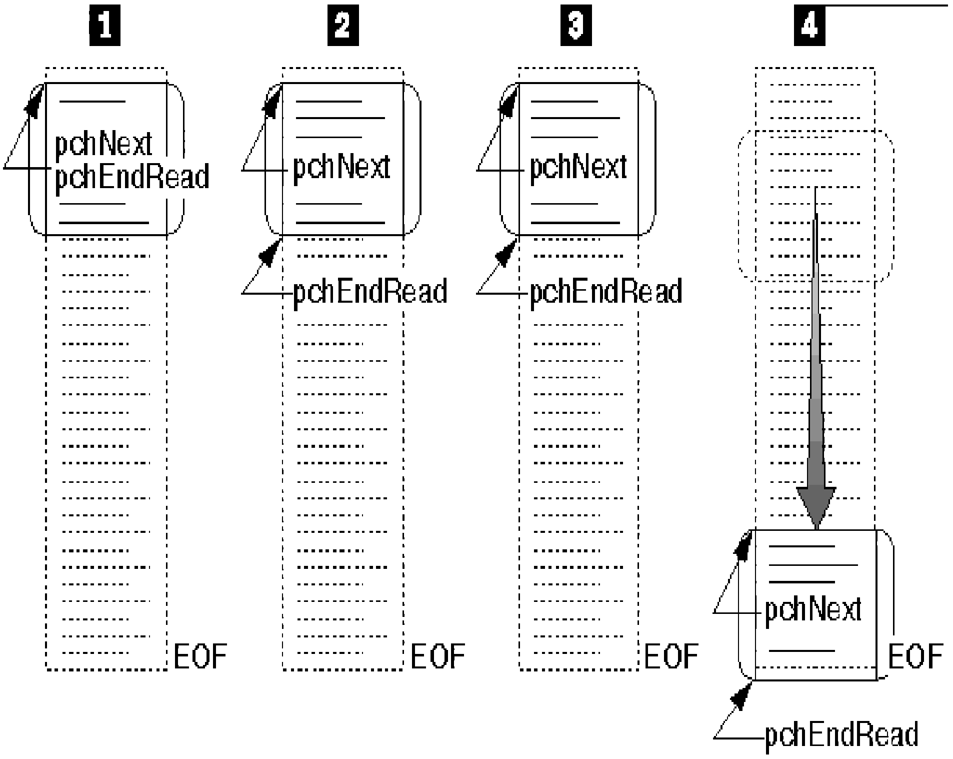
Advancing the File I/O Buffer

When you reach the end of the file I/O buffer, use mmioAdvance to *advance* the buffer. Advancing the buffer allows you to fill a file I/O buffer from disk (MMIO_READ). If there is not enough data remaining in the file to fill the buffer, the *pchEndRead* field in the MMIOINFO structure points to the location following the last valid byte in the buffer.

mmioAdvance also allows you to empty the contents of the current buffer to disk (MMIO_WRITE) by setting the MMIO_DIRTY flag in the *ulFlags* field of the MMIOINFO structure. mmioAdvance updates the fields in the MMIOINFO structure to reflect the new state of the I/O buffer (including *pchNext* , *pchEndRead* , and *pchEndWrite*).

Advancing a File I/O Buffer for Reading

The following figure shows how the file I/O buffer is advanced as a file is read from.



(1) The application opens the file for buffered I/O. The buffer is initially empty, so mmioOpen sets *pchNext* and *pchEndRead* to point to the beginning of the file I/O buffer.

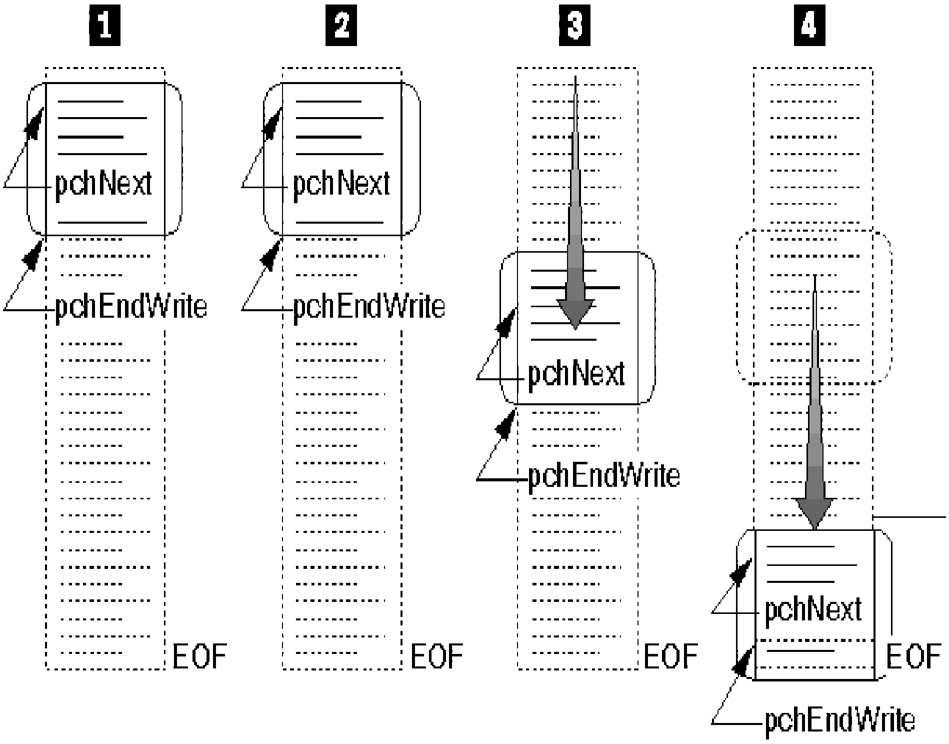
(2) The application calls mmioAdvance to fill the I/O buffer. The mmioAdvance function fills the buffer and sets *pchNext* to point to the beginning of the buffer.

(3) The application reads from the I/O buffer and increments *pchNext* .

(4) The application continues to read the buffer and call mmioAdvance to refill the buffer when it is empty. When mmioAdvance reaches the end of the file, there is not enough information to fill the buffer. The mmioAdvance function sets *pchEndRead* to point to the end of the valid data in the buffer.

Advancing a File I/O Buffer for Writing

The following figure shows how the file I/O buffer is advanced as a file is written to.



(1) The application opens the file for buffered I/O by calling mmioOpen. The mmioOpen function sets *pchNext* to point to the beginning of the file I/O buffer and *pchEndWrite* to point to the end of the buffer.

(2) The application writes to the I/O buffer and increments *pchNext* .

(3) Once the application fills the buffer, it calls mmioAdvance to empty the contents of the buffer to disk. The mmioAdvance function resets *pchNext* to point to the beginning of the buffer.

(4) The application continues to write to the buffer and call mmioAdvance to empty the contents of the buffer when its full. At the end of the file, there is not enough information to fill the buffer. When the application calls mmioAdvance to empty the contents of the buffer, *pchNext* points to the end of the valid data in the buffer.

Ending Direct Access of a File I/O Buffer

When you finish accessing a file I/O buffer, pass the MMIOINFO structure filled by mmioGetInfo to mmioSetInfo to end direct access to the I/O buffer. Before calling mmioSetInfo, make sure that you set the MMIO_DIRTY flag of the *ulFlags* field of *pmmioinfo* if you have written to the buffer. Otherwise, the contents of the buffer will not get emptied to disk. When mmioSetInfo is called, then the caller should stop accessing the I/O buffer directly and revert to using mmioRead and mmioWrite to read from and write to the file.

The following code fragment illustrates how to directly read an I/O buffer.

```
mmioGetInfo(hmmio, &mmioinfo, 0)
mmioAdvance(hmmio, &mmioinfo, MMIO_READ)
for (i=0, iCount=0;  i<20; i++)
 iCount += *(mmioinfo.pchNext)++;
mmioSetInfo(hmmio, &mmioinfo, 0);
```

File I/O in Memory

A memory file is a block of memory that is perceived as a file by an application. This can be useful if you already have a file image in memory. Memory files let you reduce the number of special-case conditions in your code because, for I/O purposes, you can treat file memory images as if they were disk-based files.

Like I/O buffers, memory files can use memory allocated by the application or by the MMIO Manager. In addition, memory files can be expandable or non-expandable.

Memory is expandable when the system allocates an internal buffer using the MMIO_ALLOCBUF flag of the mmioOpen function. When the MMIO Manager reaches the end of an expandable memory file, it expands the memory file by a predefined increment.

Use the mmioOpen function to open a memory file. Specify NULL for the *szFileName* parameter and the MMIO_READWRITE flag, as shown:

```
hmmio = mmioOpen(NULL, &mmioinfo, MMIO_READWRITE);
```

In addition, set the *pmmioinfo* parameter to point to an MMIOINFO structure set up as follows:

- Set the *pIOProc* field to NULL.
- Set the *fccIOProc* field to FOURCC_MEM.
- Set the *pchBuffer* field to point to the memory block. To request that the MMIO Manager allocate the memory block, set *pchBuffer* to NULL.
- Set the *cchBuffer* field to the initial size of the memory block.
- Set the *aulInfo* [0] field to the minimum expansion size of the memory block. For a non-expandable memory file, set *aulInfo* [0] to NULL.
- Set all other fields to 0.

The following code fragment shows how to open a memory file using a buffer named achMyBuffer.

```
        /* set mmioinfo structure to 0 */
        mmioinfo.fccIOProc= FOURCC_MEM
        mmioinfo.pchBuffer= achMyBuffer
        mmioinfo.cchBuffer= cchMyBuffer
hmmio = mmioOpen("NULL", &mmioinfo, 0);
```

The following code fragment shows how to open a memory file with 1 byte initially and expand up to 1KB as required.

```
        /* set mmioinfo structure to 0 */
        mmioinfo.fccIOProc= FOURCC_MEM
        mmioinfo.pchBuffer= NULL
        mmioinfo.cchBuffer= 1
        mmioinfo.aulInfo[0] = 1024;
hmmio = mmioOpen("NULL", &mmioinfo, MMIO_CREATE)
```

Allocating Memory for Memory Files

There are no restrictions on allocating memory for use as a non-expandable memory file. You can use static memory or stack memory, or you can use locally allocated or globally allocated memory.

Resource Interchange File Format (RIFF) Services

The Resource Interchange File Format (RIFF) is the standard file format used for storing multimedia files. RIFF enables audio, image, animation, and other multimedia elements to be stored in a common format. RIFF is also used as the basis for defining new file formats for OS/2 multimedia software.

RIFF file I/O provides simple functions to locate, create, enter, exit, and access the RIFF *chunk*-the basic building block of a RIFF file. You can open, read from, and write to RIFF files the same way as other file types. Blocks of data are identified by tags. An advantage of tagged file formats is that an application can process blocks that it understands while ignoring blocks that do not concern it. RIFF can also be expanded upon (by adding new tags) without breaking existing applications.

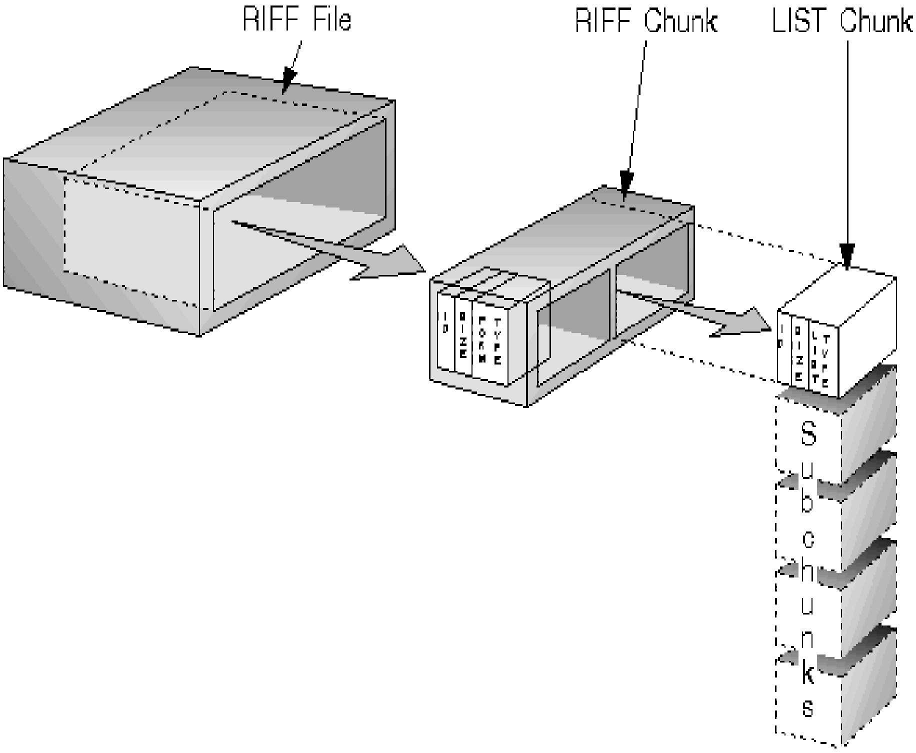
A RIFF file created with mmioOpen can hold a single data object or, if it is built as a compound file, multiple data objects. Data objects in a compound file are referred to as chunks. Chunks in a compound file are its table of contents, and the multiple data objects stored in the resource group.

Note: Refer to the *OS/2 Multimedia Programming Reference* for detailed information on how to define an application using the RIFF tagged file structure.

RIFF File Structural Overview

A RIFF chunk begins with a chunk ID, which is a four-character code (FOURCC) that identifies the representation of the chunk data. A program reading a RIFF file can skip over any chunk whose chunk ID it does not recognize. The chunk ID is followed by a four-character chunk size (ULONG) specifying the size of the data field in the chunk. Lastly, it contains a data field containing the actual data of the chunk. If the chunk ID is `RIFF`, the first four characters of the data portion of the chunk are a form type; if the chunk ID is `LIST`, the first four characters are a list type.

The only chunks allowed to contain other chunks (subchunks) are those with a chunk ID of `RIFF` or `LIST`. The first chunk in a RIFF file must be a RIFF chunk. All other chunks in the file are subchunks of the RIFF chunk as shown.



RIFF Chunks

RIFF chunks include an additional field in the first 4 bytes of the data field. This additional field provides the *form type* of the field, which is a four-character code identifying the format of the data stored in the file. A RIFF form is simply a chunk with a chunk ID of RIFF. For example, waveform audio files (WAVE files) have a form type of WAVE.

LIST Chunks

A LIST chunk contains a list, or ordered sequence, of subchunks. LIST chunks also include an additional field in the first 4 bytes of the data field. This additional field contains the *list type* of the field, which is a four-character code identifying the contents of the list. For example, a LIST chunk with a list type of INFO can contain ICOP and ICRD chunks providing copyright and creation date information.

If an application recognizes the list type, it should know how to interpret the sequence of subchunks. However, since a LIST chunk may contain only subchunks (after the list type), an application that does not know about a specific list type can still navigate through the sequence of subchunks.

RIFF File Functions

The following MMIO functions enable you to manage RIFF files:

Function	Description
mmioFOURCC	Converts four characters into a four-character code (FOURCC).
mmioStringToFOURCC	Converts a null-terminated string into a four-character code.
mmioCreateChunk	Creates a chunk in a RIFF file that was opened by mmioOpen.
mmioAscend	Ascends out of a chunk in a RIFF file that was descended into by mmioDescend or created by mmioCreateChunk.
mmioDescend	Descends into a RIFF file chunk beginning at the current file position, or searches for a specified chunk.

The MMCKINFO Data Structure

Several multimedia file I/O functions use the MMCKINFO structure to specify and retrieve information about a chunk in a RIFF file. The MMIOOS2.H header file defines the MMCKINFO structure as as shown.

```
typedef struct _MMCKINFO {   /* mmckinfo                              */
  FOURCC      ckid;          /* Chunk id (FOURCC)                     */
  ULONG       ulSize;        /* Chunk size (bytes)                    */
  FOURCC      fccType;       /* FOURCC type (if ckid RIFF or LIST)    */
  ULONG       ulDataOffset;  /* File offset of data portion of chunk  */
  ULONG       ulFlags;       /* MMIO_DIRTY (if new chunk)             */
} MMCKINFO;
```

Four-Character Codes

A four-character code is a 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters. The data type for a four-character code is FOURCC. Use the mmioFOURCC function to convert four characters to a four-character code. For example, to use a four-character code for WAVE:

```
FOURCC   fccIOProc;

fccIOProc = mmioFOURCC( 'W', 'A', 'V', 'E' ) ;
```

To convert a null-terminated string into a four-character code, use the mmioStringToFOURCC function. The following example also generates a four-character code for WAVE.

```
FOURCC fccIOProc;

fccIOProc = mmioStringToFOURCC("WAVE", 0);
```

The second parameter in mmioStringToFOURCC specifies options for converting the string to a four-character code. If you specify the MMIO_TOUPPER flag, mmioStringToFOURCC converts all alphabetic characters in the string to uppercase. This is useful when you need to specify a four-character code to identify a custom I/O procedure. (Four-character codes are case-sensitive.)

Creating RIFF Chunks

Use the mmioCreateChunk function to create a new chunk by writing a chunk header at the current position in an open file and then "descending" into the chunk. (See Descending into a Chunk for further information.) You must specify a pointer to a MMCKINFO structure containing information about the new chunk. You also need to determine which chunk type to create by specifying MMIO_CREATERIFF or MMIO_CREATELIST. The return value is 0 if the chunk is successfully created; otherwise, the return value specifies an error code.

The following code fragment illustrates how to create a new chunk with a chunk ID of `RIFF` and the form type of `WAVE`.

```
HMMIO         hmmio;
MMCKINFO      mmckinfo;
.
.
.
mmckinfo.fccType = mmioFOURCC('W', 'A', 'V', 'E');
mmioCreateChunk(hmmio, mmckinfo, MMIO_CREATERIFF);
```

If you are creating a RIFF or LIST chunk, you must specify the form type in the *fccType* field of the MMCKINFO structure. In the previous example, the form type is `WAVE`.

If you know the size of the data field in the new chunk, set the *ckSize* field in the MMCKINFO structure when you create the chunk. This value is written to the *ckSize* field in the new chunk. If this value is not correct when you call mmioAscend to mark the end of the chunk, it is automatically rewritten to reflect the correct size of the data field.

After you create a new chunk using mmioCreateChunk, the file position is set to the data field of the chunk (8 bytes from the beginning of the chunk). If the chunk is a RIFF or LIST chunk, the file position is set to the location following the form type or list type (12 bytes from the beginning of the chunk). The *ckSize* field is assumed to be a "proposed chunk size" if it turns out to be correct (if you write that much data into the chunk before calling mmioAscend to end the chunk, the mmioAscend will not have to seek back and correct the chunk header.)

Moving between Chunks

RIFF files may consist of nested chunks of information. MMIO services include two functions you can use to move between chunks in a RIFF file: mmioAscend and mmioDescend. You might think of these functions as high-level seek functions. When you descend into a chunk, the file position is set to the data field of the chunk (8 bytes from the beginning of the chunk). For RIFF and LIST chunks, the file position is set to the location following the form type or list type (12 bytes from the beginning of the chunk). When you ascend out of a chunk, the file position is set to the location following the end of the chunk.

Descending into a Chunk

The mmioDescend function descends into a chunk or searches for a chunk, beginning at the current file position. The mmioDescend function requires a *pckinfo* parameter, which specifies a pointer to a MMCKINFO structure that mmioDescend fills with information on the current chunk. You can also specify the *pckinfoParent* parameter, which specifies an optional caller-supplied structure that refers to the parent of the chunk that is being searched for. If there is no parent chunk, set *pckinfoParent* to NULL.

The *usFlags* parameter specifies options for searching for a chunk. Choose from the following options:

Flag	Description
MMIO_FINDCHUNK	Searches for a chunk with a specific chunk ID. The *ckid* field *pckinfo* should contain the chunk ID of the chunk to search for when mmioDescend is called.
MMIO_FINDRIFF	Searches for a chunk with a RIFF chunk ID and with a specific form type. The *fccType* field of *pckinfo* should contain the form type of the RIFF chunk to search for when mmioDescend is called.
MMIO_FINDLIST	Searches for a chunk with a chunk ID of LIST and with a specific list type. The *fccType* field of *pckinfo* should contain the list type of the LIST chunk to search for when mmioDescend is called.

Note: If you do not specify any flags, mmioDescend descends into the chunk that starts at the current file position.

The mmioDescend function fills an MMCKINFO structure with information on the chunk. This information includes the chunk ID (*ckid*), the size of the data field (*ckSize*), and the form type, or list type, depending on whether the chunk is a RIFF or LIST chunk. The mmioDescend function assumes that the current file position is the beginning of a chunk header when mmioDescend is called. If *pckinfoParent* is given, mmioDescend assumes that the current file position is within *pckinfoParent* (a RIFF or LIST chunk).

Searching for a Chunk

To search for a chunk in an open RIFF file, use mmioDescend with the MMIO_FINDCHUNK parameter. Also set the *ckid* field of the MMCKINFO structure referenced by *pckinfo* to the four-character code of the chunk you want to search for.

If you are searching for a RIFF or LIST chunk, the mmioDescend function sets the *ckid* field of the MMCKINFO structure. Set the *fccType* field to the four-character code of the form type or list type of the chunk.

Ascending out of a Chunk

After you descend into a chunk and read the data in the chunk, you can move the file position (pointer) to the beginning of the next chunk. This is accomplished by ascending out of the chunk using the mmioAscend function. The mmioAscend function specifies a pointer to an MMCKINFO structure identifying a chunk. The function ascends to the location following the end of this chunk. The return value is 0 if the operation is successful; otherwise, the return value specifies an error code.

If the chunk was descended into using mmioDescend, then mmioAscend seeks to the location following the end of the chunk (past the extra pad byte, if any).

If the chunk was created and descended into using mmioCreateChunk (the MMIO_DIRTY flag in the *ulFlags* field of *pckinfo* is set), then the current file position is assumed to mark the end of the data portion of the chunk. If the chunk size is not the same as the value that was stored in *ckSize* of *pckinfo* before mmioCreateChunk was called, then mmioAscend seeks back and corrects the chunk size in the chunk header before ascending from the chunk. Also, if the chunk size is odd, then mmioAscend writes a null pad byte at the end of the chunk.

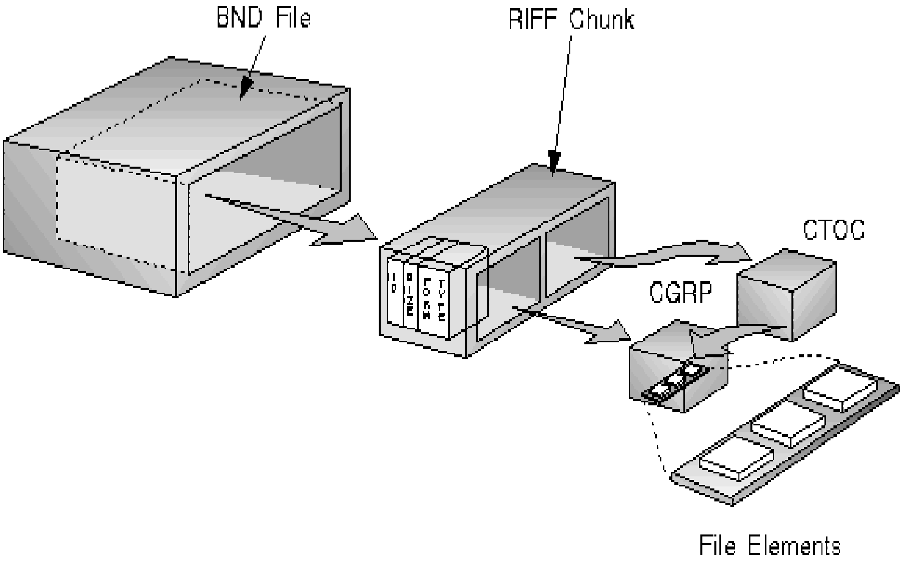
RIFF Compound File Overview

Files based upon the compound file structure contain the following two RIFF chunks at the "top level" of a RIFF file-(as subchunks of the RIFF chunk):

- Compound File Resource Group (CGRP) chunk
- Compound File Table of Contents (CTOC) chunk

The CGRP chunk contains all the compound file elements, concatenated together. An element may be a RIFF file, but it may also be a non-RIFF file, or an arbitrary RIFF chunk, or arbitrary binary data. The definition of the form that contains the CGRP chunk may specify exactly what the elements of the CGRP chunk may be. The CTOC chunk indexes the CGRP chunk, which contains the actual multimedia data elements. Each entry contains the name of the element and other information about the element, including the offset of the element within the CGRP chunk. All the CTOC entries of a table are of the same length and can be specified when the file is created.

The CTOC chunk may appear either before or after the CGRP chunk. Generally, the CTOC chunk is placed at the front of the file to reduce the seek and read times required to access it. See the following figure.



RIFF Compound File Functions

A RIFF compound file can contain multiple file elements. A file element is an individual file that is part of a RIFF compound file. An element of a compound file also could be an entire RIFF file. The MMIO Manager provides service to find, query, and access any file elements in a compound file. It also supports the function of file compaction.

The following MMIO functions enable you to manage RIFF compound files:

Function	Description
mmioCFOpen	Opens a RIFF compound file by

```
                               name.

mmioCFClose                    Closes a RIFF compound file
                               that was opened by mmioCFOpen.

mmioCFGetInfo                  Retrieves the CTOC header of
                               an open RIFF compound file.

mmioCFSetInfo                  Modifies information that is
                               stored in the CTOC header of
                               an open RIFF compound file.

mmioCFAddEntry                 Adds an entry to the CTOC
                               chunk of an open RIFF compound
                               file.

mmioCFChangeEntry              Changes a CTOC entry in an
                               open RIFF compound file.

mmioCFFindEntry                Finds a CTOC entry in an open
                               RIFF compound file.

mmioCFDeleteEntry              Deletes a CTOC entry in an
                               open RIFF compound file.

mmioCFAddElement               Adds an element to the CGRP
                               chunk of an open RIFF compound
                               file.

mmioCFCopy                     Copies the CTOC and CGRP
                               chunks from an open RIFF
                               compound file to another RIFF
                               compound file.

mmioCFCompact                  Compacts a RIFF compound file
                               by removing elements marked as
                               deleted.

mmioFindElement                Enumerates the entries of a
                               compound file.

mmioRemoveElement              Removes the specified element
                               in a compound file.
```

The MMCFINFO Structure

The MMIO Manager uses the MMCFINFO data structure to maintain state information on an open file. The MMCFINFO data structure is defined in the MMIOOS2.H header file as shown.

```
typedef struct _MMCFINFO  /* mmcfinfo                                   */
 {
 ULONG  ulHeaderSize;     /* CTOC header size                            */
 ULONG  ulEntriesTotal;   /* Num of CTOC table entries                   */
 ULONG  ulEntriesDeleted; /* Num of CTOC table entries to deleted CGRP   */
 ULONG  ulEntriesUnused;  /* Num of unused CTOC entries                  */
 ULONG  ulBytesTotal;     /* Combined byte size of all CGRP elements     */
 ULONG  ulBytesDeleted;   /* Byte size of all deleted CGRP elements      */
 ULONG  ulHeaderFlags;    /* Information about entire compound file (CF) */
 USHORT usEntrySize;      /* Size of each CTOC table entry               */
 USHORT usNameSize;       /* Size of name field in entry, default 13     */
 USHORT usExHdrFields;    /* Num CTOC header extra fields                */
 USHORT usExEntFields;    /* Num CTOC entry extra fields                 */
 } MMCFINFO;
```

Opening or Creating a RIFF Compound File

To perform I/O procedures on a new or existing RIFF compound file, an application issues the mmioCFOpen function. This function constructs a CTOC in memory for a RIFF compound file. Specify the *pmmiocfinfo* parameter to identify a pointer to a user-supplied CTOC header structure containing optional header information. The *pmmiocfinfo* parameter can be NULL if the default values of the fields are sufficient. You can also specify the *pmmioinfo* parameter, which identifies a pointer to a user-supplied info structure containing optional open information that is passed to mmioOpen.

The *ulFlags* options for mmioCFOpen include the following.

Flag	Description
MMIO_READ	Opens a file for reading only (default).
MMIO_WRITE	Opens a file for writing only.
MMIO_READWRITE	Opens a file for both reading and writing.
MMIO_CREATE	Creates a new file.
MMIO_EXCLUSIVE	Opens a file with exclusive mode, denying other processes both read and write access to the file.
MMIO_DENYWRITE	Opens a file and denies other processes write access to the file.
MMIO_DENYREAD	Opens a file and denies other processes read access to the file.
MMIO_DENYNONE	Opens a file without denying other processes read or write access to the file.

If the file does not exist, an error is returned unless you specified the MMIO_CREATE option. If the file exists but is not a RIFF compound file, the system returns an error, regardless of whether or not you specified MMIO_CREATE.

The access and sharing flags are maintained only within the set of compound file (CF) functions. If the RIFF compound file or elements are accessed without using the CF functions, the access and sharing modes are unpredictable. An mmioOpen function with a fully qualified element name is considered a CF function since it internally calls mmioCFOpen, thus the flags are predictable in that case.

Closing a RIFF Compound File

Use the mmioCFClose function to close a RIFF compound file that was opened by mmioCFOpen. The mmioCFClose function writes the CTOC back to the RIFF compound file. You can also open an element using the mmioOpen function with the BND and element both specified on the open call (see to Opening or Creating a File). In that case, you would call the mmioClose function, which would close the element and RIFF compound file.

If the process ends, all open elements are closed and the CTOC is rewritten. If the compound file was opened for read only, the CTOC is not rewritten.

If the mmioCFClose fails and you modified CGRP elements, the data stored on the file is inconsistent. Attempt to correct the inconsistency by freeing file space and trying to close the file again.

Retrieving Information

Use the mmioCFGetInfo function to retrieve the CTOC header of an open RIFF compound file. The mmioCFGetInfo function requires a *pmmcfinfo* parameter which identifies a pointer to a user-supplied buffer that will be filled with the CTOC header. Use the *cBytes* parameter to specify the size of the *pmmcfinfo* buffer. This is the maximum number of bytes that will be copied.

The information copied to *pmmcfinfo* consists of a MMCFINFO structure followed by variable length arrays *aulExHdrFldUsage* , *aulExEntFldUsage* , and *aulExHdrField* .

To find out how large a buffer the user needs to allocate, call mmioCFGetInfo with *cBytes* equal to the size of a ULONG. This returns the first field of the CTOC header, which happens to be the size of the header. This size can then be used as *cBytes* on the subsequent call.

Modifying the CTOC Header

Use the mmioCFSetInfo function to modify information that is stored in the CTOC header of an open RIFF compound file. You should only modify the *aulExHdrFldUsage* and *aulExHdrField* fields.

The mmioCFSetInfo function requires a *pmmcfinfo* parameter which identifies a pointer to a user-supplied buffer that contains the modified CTOC header. This buffer was filled in by mmioCFGetInfo and then modified by the user. Use the *cBytes* parameter to specify the size of the *pmmcfinfo* buffer. This is the maximum number of bytes that will be copied.

The MMCTOCENTRY Structure

The MMIO Manager uses the MMCTOCENTRY data structure to maintain state information on an open file. The MMCTOCENTRY data structure is defined in the MMIOOS2.H header file as shown.

```
typedef struct _MMCTOCENTRY  {
 ULONG    ulOffset;         /* Offset of element within CGRP       */
 ULONG    ulSize;           /* Size of element                     */
 ULONG    ulMedType;        /* FOURCC of element                   */
 ULONG    ulMedUsage;       /* Possible sub type                   */
 ULONG    ulCompressTech;   /* Compression technique used          */
 ULONG    ulUncompressBytes;/* Actual size of uncompressed element */
 } MMCTOCENTRY;
```

Adding an Entry to the CTOC Chunk

Use the mmioCFAddEntry function to add an entry to the CTOC chunk of an open RIFF compound file. (Do not duplicate entries.) The mmioCFAddEntry function requires a *pmmctocentry* parameter which identifies a pointer to a user-supplied CTOC structure containing the CTOC data. The "identifier" for the entry is the element name, which is passed in the *pmmctocentry* buffer. If mmioCFAddEntry expands the current number of entries past the number currently allocated, on a mmioCFClose the CTOC is written following the CGRP in the file.

Changing a CTOC Entry

Use the mmioCFChangeEntry function to modify a CTOC entry in an open RIFF compound file. The mmioCFChangeEntry function requires a *pmmctocentry* parameter which identifies a pointer to a user-supplied CTOC structure containing modified CTOC data. The "identifier" for the entry is the element name, which is passed in the *pmmctocentry* buffer. The mmioCFChangeEntry function updates the CTOC entry with the information contained in the user's *pmmctocentry* . If you change the compression technique, you must also modify the *ulUncompressBytes* field. When the compression technique is NULL, the uncompressed bytes field must be the size in bytes of the element when it is uncompressed.

Finding a CTOC Entry

The mmioCFFindEntry function enables you to find a particular entry in an open RIFF compound file. The mmioCFFindEntry function requires the *pmmctocentry* parameter which identifies a pointer to a user-supplied CTOC structure containing the name of the RIFF compound file element to search for. You can set flags in *ulFlags* to specify that an element is to be searched for by some attribute other than its name.

Flags for mmioCFFindEntry include the following.

Flag	Description
MMIO_FINDFIRST	Finds the first entry in the CTOC table. Note: MMIO_FINDFIRST is ignored if you set either MMIO_FINDDELETED or MMIO_FINDNEXT.
MMIO_FINDNEXT	Finds the next entry in the CTOC table after the entry that contains the element searched for.
MMIO_FINDDELETED	Finds the first entry in the table that has been marked as "deleted", or the next deleted entry following the entry that contains the element to search for.

The search is case-insensitive. If no flags are set, the search is for the element only. If the function succeeds, the *pmmctocentry* buffer is filled with information about the CTOC entry. You can progress through the CTOC entry list by doing a FINDFIRST followed by a series of FINDNEXT, using the information from the previous function.

Deleting a CTOC Entry

Use the mmioCFDeleteEntry function to delete a CTOC entry in an open RIFF compound file. The mmioCFDeleteEntry function requires a *pmmctocentry* parameter, which identifies a pointer to a user-supplied CTOC structure containing the RIFF compound file element name. The "identifier" for the entry is the element name, which is passed in the *pmmctocentry* field. The entry is marked "deleted" by the FOURCC of FOURCC_DEL. The actual element data remains in place. To physically remove both the entry and the element's data, use the mmioCFCopy function.

Adding an Element to the CGRP Chunk

Use mmioCFAddElement to add an element to the CGRP chunk of an open RIFF compound file. The mmioCFAddElement function requires:

- A pointer to the name of the element that you want to add to the CGRP chunk (*pszElementName*)
- The four-character code of the element (*fccType*)
- A pointer to the caller-supplied buffer containing the element data (*pchBuffer*)
- The size of the caller-supplied buffer (*cchBytes*).

The CTOC entry for the element does not have to exist before you call mmioCFAddElement. If the CTOC entry exists, mmioCFChangeEntry modifies its contents. If the CTOC entry does not exist, mmioCFAddEntry is called to add the CTOC entry for this element. The mmioCFAddElement function writes the element to the end of the CGRP chunk. The user's buffer contains the element data.

The CGRP chunk may precede the CTOC chunk and overwrite the CTOC on the file system. This is corrected when the RIFF compound file is closed and the CTOC is rewritten.

Note: You can also add an element to the CGRP chunk by specifying MMIO_CREATE using the mmioOpen function.

Copying CTOC and CGRP Chunks

Use the mmioCFCopy function to copy the CTOC and CGRP chunks from an open RIFF compound file to another RIFF compound file. The mmioCFCopy function requires a *pszDestFileName* parameter, which identifies the pointer to the name of the destination file.

The mmioCFCopy function opens the destination file for MMIO_CREATE (using mmioOpen) and builds a RIFF BND header at the beginning of the file. The CTOC and CGRP chunks are then copied. The newly written CGRP chunk is compacted; it has no deleted elements.

Compacting RIFF Compound Files

Use the mmioCFCompact function to compact a RIFF compound file in place. The file must not be opened by any other process or the compaction fails. Upon success a new CGRP is written into the same source file with no deleted elements.

Sample Application Programs

This section gives a brief overview of the sample application programs provided with the Toolkit, including programming concepts and program flow diagrams for each of the sample programs.

The purpose of the OS/2 multimedia sample programs is to illustrate multimedia programming concepts and establish a basis for creating your own multimedia applications. The programs provide you with practical examples that span a range of multimedia concepts. The source code provided shows you how to use multimedia controls and functions to create multimedia applications. Each sample program serves as a template that can be modified easily to meet your multimedia application requirements. The samples were compiled and verified using the IBM C Set ++ and VisualAge C++ compilers.

Some samples require specific hardware devices. Without these devices, you can still compile and run the sample programs; however, you might not receive the full effect of the program. For example, if a sample program has audio, you will not hear it unless you have a supported audio adapter and speakers installed.

Subdirectory Structure

Code for the sample programs (and associated files such as waveform audio files and movie files) are located in the following subdirectories.

```
 TOOLKIT   OS/2 Developer's Toolkit


          H      Header Files
```

```
  INC     Include Files

  LIB     Library Files

SAMPLES   Samples

         MM     OS/2 Multimedia Samples

            ASYMREC   Asymmetric Recording Sample

            AVCINST   AVC I/O Procedure Installation Sample

             CAPDLL   Caption DLL

            CAPSAMP   Caption Sample Application

            CAPTION   Caption Creation Utility

             CLOCK    Memory Playlist Sample

            DOUBPLAY  Double Buffering Playlist Sample

             DIVE     Direct Interface Video Extensions Sample

             DUET1    Streaming Device Duet Sample

             DUET2    Streaming and Non-Streaming Device Duet Sample

             MCISPY   MCISpy Sample

            MCISTRNG  Media Control Interface String Test Sample

            MMBROWSE  Image Browser Sample

             MOVIE    Movie Sample

            RECORDER  Audio Recorder Sample

            SHORTCF   Control File Templates

             TUNER    TV Tuner Sample

            ULTIEYES  Non-Linear Video Sample
```

Note: The SHORTCF subdirectory contains control file templates you can utilize when installing a program using MINSTALL. See Installing a Program Using MINSTALL for further details.

ASYMREC - Asymmetric Recording Sample

This sample (ASYMREC) illustrates how to include asymmetric recording function in your multimedia application. Modules include source code extracted from the Video IN Recorder application, which enables frame-step recording using Ultimotion compression techniques.

Frame-step recording captures the audio and video in a two-pass operation. First, it captures the audio in real time. It then returns to the point in the media where recording started to capture the video frame by frame.

Source Code

The following files are located in the \TOOLKIT\SAMPLES\MM\ASYMREC subdirectory.

ASYMREC.H
Is the header file for the asymmetric recording sample. See this file for a complete description of the asymmetric recorder data structures.

ASYMREC.C
Is the sample program code for the asymmetric recording sample. It includes code for the following functions:

OpenMMIO
Opens and identifies the I/O procedure to install.

IdentifyIOProc
Loads and installs the known I/O procedure. Also, attempts to open the movie file using this I/O procedure.

AssociateCodec
Sends MMIOM_SET to associate the CODEC for later use in compression.

InitiateFrameStepRecord
Initializes the environment for frame-step recording and starts a frame-step record thread.

CompressBuffer
Calls the AVI I/O procedure and Ultimotion CODEC procedure to compress a buffer.

StartVideoThread
Starts a recording thread for doing the frame-step.

Detailed information about each function is provided in the source code.

Real-Time Capture and Asymmetric Capture

Typically, a stream is established to read a series of moving images from the device. This continuous capture mode is called symmetric or *real-time* capture, and is performed at a constant frame rate and constant frame size. When established, a stream captures images continuously without assistance from the application code. However, video images can be captured one frame at a time when directed by the application. This type of capture mode is referred to as *asymmetric* capture because the time between successive frames is not constant. The frame rate is determined by how long the application delays between taking image snapshots. Asymmetric capture is performed with IOCtls only; the OS/2 multimedia streaming mechanism is not used.

Asymmetric Recording Architecture

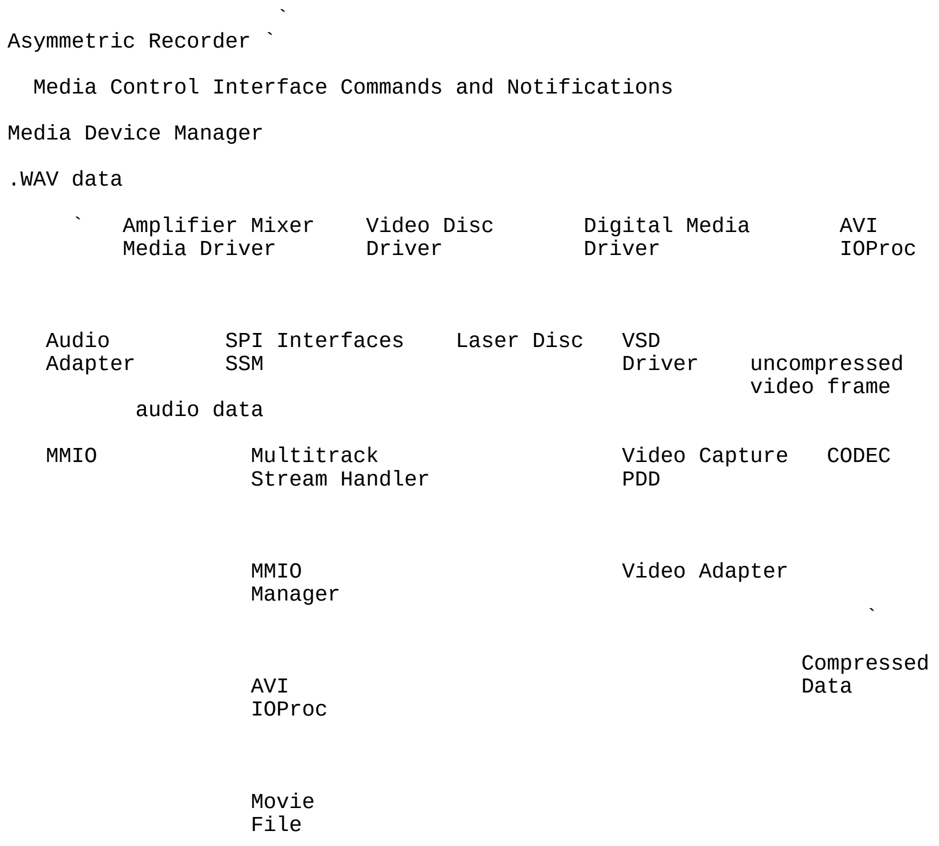
Several passes through the source data are required to produce an Ultimotion movie file.

During the first pass, the capture routines position the video source device to the starting position in the source video. It then uses OS/2 multimedia to record the desired audio track. If the desired final format for the movie requires both audio and video to be interleaved, MULTITRACKWRITE interleaves them.

During the second pass, capture routines use OS/2 multimedia to position the video source at the desired location. For each frame in the movie, an MCI_STEP message is sent to the MCD controlling the source device. When the resulting image has been "grabbed" or "digitized" by the capture hardware, the video data is retrieved and written to disk through the AVI I/O procedure. This process continues until the required video is captured.

Note: Only the second pass is used if you are recording video without audio.

The following figure illustrates the architecture of the components involved in doing asymmetric capture and compression of video from frame accurate devices.



MMMULTITRACKREAD Operations

The MMMULTITRACKREAD structure contains the following parameters:

- Pointer to the read buffer (*pBuffer*)
- Length of the read buffer (*ulLength*)
- Read flags (*ulFlags*)
- Number of track entries or number of tracks (*ulNumTracks*)
- Pointer to a track map list (*pTrackMapList*)
- Actual buffer length available in MULTITRACKREAD extended mode (*ulBufferLength*)

The track map list is a list of valid track numbers for the current file and is used to map a track to a record table. Each track map list contains:

- A track ID (*ulTrackID*)
- The number of record entries (*ulNumEntries*)

- A pointer to a record table (*pRecordTabList*)

Each element of the record table contains a pointer to (*pBuffer*) and the length (*ulLength*) of the corresponding data in the buffer.

For example, if the number of track entries is two, the track map list will contain two track map tables, one for each track. There is a corresponding record table for each track. The number of entries in the track map table for each track is the number of entries in the record table for a particular track. The following figure illustrates the multitrack read data structure.

```
  MMMULTITRACKREAD
                                                pBuffer

  ulLength           TRACKMAP         RECORDTAB
  pBuffer            TABLE                  AUDIO
  ulFlags
  ulNumTracks        ulTrackID          pRecord  1        Audio
  pTrackMapList      ulNumEntries       ulLength          Data 1
                     pRecordTabList     ulParm1
                                        ulParm2           Video
  ulBufferLength     ulTrackID                            Data 1
 (extended mode)     ulNumEntries       .        2
                     pRecordTabList     .                 Audio
                                        .                 Data 2
                     .
                     .                RECORDTAB           Video
                     .                      VIDEO         Data 2     ulLength

                                        pRecord  1        .
                                        ulLength          .
                                        ulParm1           .
                                        ulParm2

                                        .        2
                                        .
                                        .
```

```
MMMULTITRACKREAD                      pBuffer

  ulLength           TRACKMAP         RECORDTAB
  pBuffer            TABLE                  AUDIO
  ulFlags
  ulNumTracks        ulTrackID          pRecord  1        Audio
  pTrackMapList      ulNumEntries       ulLength          Data 1
                     pRecordTabList     ulParm1
                                        ulParm2           Video
  ulBufferLength     ulTrackID                            Data 1
 (extended mode)     ulNumEntries       .        2                   ulLength
                     pRecordTabList     .                 Audio
                                        .                 Data 2
                     .
                     .                RECORDTAB           Video
                     .                      VIDEO         Data 2

                                        pRecord  1        .
                                        ulLength          .
                                        ulParm1           .
                                        ulParm2

                                        .        2
                                        .
                                        .
```

Multitrack Reading

MULTITRACKREAD is supplied an empty buffer. The size is determined by *ulLength*, and it is pointed to by *pBuffer*. MULTITRACKREAD processes the data by reading *ulLength* bytes of data into the buffer, and then parsing the data in the buffer by media type (audio or video) into records. A pointer to the data in the buffer is placed in the appropriate record table.

Note: A record is a contiguous buffer containing data of the same media type.

During the processing of MULTITRACKREAD in extended mode, *pBuffer* points to the beginning of the buffer on the first call to MULTITRACKREAD and *ulLength* indicates the number of bytes to be read into the buffer. With each call to MULTITRACKREAD, *ulBufferLength* (the actual total buffer length) is reduced by *ulLength* bytes, and *pBuffer* points to the current location in the buffer. Subsequent calls to MULTITRACKREAD reads *ulLength* bytes of data into the buffer starting at *pBuffer* and repeats the process until *ulBufferLength* is reached.

This implementation of MULTITRACKREAD allows smaller amounts of data to be read at one time while allowing frames greater than the *ulLength* of the read. For example, if the program wanted to fill a 128KB buffer while doing 32KB reads, the calling sequence to MULTITRACKREAD would be (32KB, 128KB), (32KB, 96KB), (32KB, 64KB), and (32KB, 32KB) for (*ulLength*, *ulBufferLength*). With this setup, MULTITRACKREAD can span frames across the 32KB buffers and know the size of the 128KB buffer.

During MULTITRACKREAD processing in regular mode, *pBuffer* always points to the beginning of the buffer and *ulLength* indicates the number of bytes to be read into the buffer. The buffer is filled completely with each call to MULTITRACKREAD and subsequent calls, to MULTITRACKREAD, read *ulLength* bytes of data into the buffer starting at the beginning of the buffer (pointed to by *pBuffer*).

Note: If the MULTITRACKREAD_EXTENDED bit flag is set, the calling routine has passed the extended MMMULTITRACKREAD structure with the new *ulBufferLength* field in this structure.

MMMULTITRACKWRITE Operations

The MMMULTITRACKWRITE structure contains the following parameters:

- Number of tracks (*ulNumTracks*)
- A pointer to a track map list (*pTrackMapList*)
- Read flags (*ulFlags*)

The track map list is a list of valid track numbers for the current file and is used to map a track for a record table. Each track map list contains:

- A track ID (*ulTrackID*)
- Number of record entries (*ulNumEntries*)
- A pointer to a record table (*pRecordTabList*)

Each element of the record table contains a pointer (*pRecord*) to and the length (*ulLength*) of the corresponding data in the buffer.

For example, if the number of track entries is two, the track map list contains two track map tables, one for each track. There is a corresponding write record table for each track. The number of entries in the track map table for each track is the number of entries in the write record table for a particular track.

Note: A record is a contiguous buffer containing data of the same media type.

The following figure illustrates the multitrack write data structure.

```
MMMULTITRACKWRITE


ulNumtracks        TRACKMAP         RECORDTAB
pTrackMapList      TABLE                  AUDIO
ulFlags
ulNumTracks        ulTrackID          pRecord  1         Audio
                   ulNumEntries       ulLength           Data 1
                   pRecordTabList     ulParm1
                                      ulParm2
                   ulTrackID                             Audio
                   ulNumEntries       .        2         Data 2
                   pRecordTabList     .
                                      .
                   .
                   .
```

```
.

                    pRecord  1         Video
                    ulLength           Data 1
                    ulParm1
                    ulParm2
                                       Video
                    .        2         Data 2
                    .
                    .
```

Multitrack Writing

MULTITRACKWRITE provides a list of records by media type and processes the data by using the pointers in the write record table to locate the data in the buffers and write the data to the file.

If the MULTITRACKWRITE_MERGE bit flag is *not* set, MULTITRACKWRITE writes all of the records sequentially from the first entry in the track map table (for the first track) followed by all records for the second track.

If the MULTITRACKWRITE_MERGE bit flag is set, MULTITRACKWRITE attempts to interleave the digitalvideo track with the digitalaudio track. Interleaving is accomplished based on the size of the audio buffer. For example, if a 4KB audio buffer represents one-third of a second and the frame rate is 15 frames per second, the interleave factor would be 5:1 (5 video frames to 1 audio chunk).

AVCINST - AVC I/O Procedure Installation Sample

This sample (AVCINST) explains how an application can install and remove an I/O procedure to use multimedia input/output (MMIO) file services. The AVC I/O Procedure Installation Sample is a simple PM application that allows you to install or deinstall the audio AVC I/O procedure, AVCAPROC.DLL.

Note: This sample shows the installation and removal of a system-defined I/O procedure. Typically, you would install a custom I/O procedure that was not built into the MMPMMMIO.INI file during installation.

Program Flow

The following figure illustrates the interaction between OS/2 multimedia system components and the AVC I/O Procedure Installation sample program. Source code is located in the \TOOLKIT\SAMPLES\MM\AVCINST subdirectory.

```
AVO I/O Procedure Installation
       Sample Program

      (1)

        MMIO Manager

      (2)

      AVC I/O Procedure
```

(1) When **OK** is selected, the AVC I/O Procedure Installation program calls the mmioInstallfIOProc function to install or deinstall the AVC I/O procedure (AVCAPROC.DLL) in the system.

(2) The MMIO Manager installs or deinstalls the AVC I/O procedure.

CAPTION - Caption Creation Utility

The Caption Creation Utility (CAPTION) is part of the sample captioning system provided with the Toolkit. See Captioning for additional information on this sample captioning system.

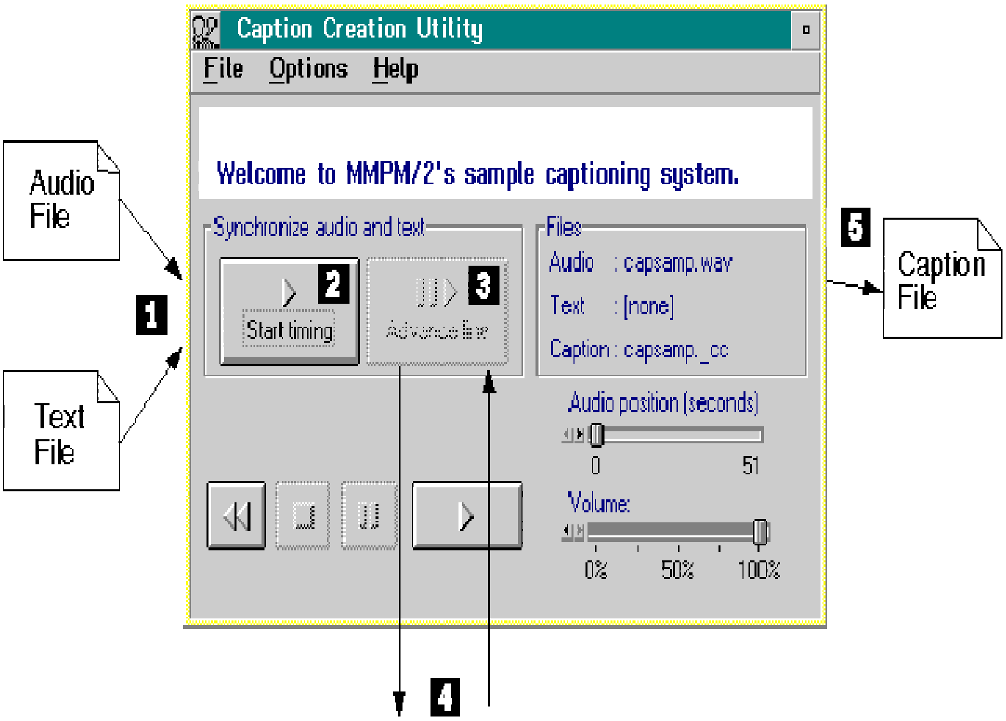
The Caption Creation Utility program enables the synchronization of an audio file with a text file.

Note: This concept can be extended beyond audio and text to apply to many possibilities, such as synchronizing audio and video, or synchronizing video and text.

While the audio file is playing, the Caption Creation Utility program issues the MCI_STATUS command to obtain a media-position value of the audio file. The media-position value is combined with the text file to produce a caption file. An application can use the resulting caption file in conjunction with the Caption DLL to provide captioning in an application.

Program Flow

The following figure illustrates the interaction between the audio, text, and caption file with the Caption Creation Utility program. Source code for the Caption Creation Utility is located in the \TOOLKIT\SAMPLES\MM\CAPTION subdirectory.



Welcome to MMPM/2's sample captioning system.

Files
Audio : capsamp.wav
Text : [none]
Caption : capsamp._cc

(1) Select an audio file and a corresponding text file to synchronize with the selected audio file. For example, you might want to synchronize the text of a poem with an audio file of someone reading the poem. When you open a text file, the first line of text appears at the bottom of the text window. The first line of the text file selected in the example shown in the previous

figure is "Welcome to MMPM/2's sample captioning system."

(2) In order to begin the synchronization process, you must select **Start timing**. The audio file begins to play and **Advance line** becomes enabled.

(3) Select **Advance line** to scroll to the next line of text. The next line of text is scrolled and appears in the text window.

(4) When you select **Advance line**, the Caption Creation Utility program passes the device ID, the MCI_STATUS command with the MCI_STATUS_ITEM flag, and the MCI_STATUS_PARMS data structure with the *ulItem* field set to MCI_STATUS_POSITION to the Media Device Manager (MDM). Upon return, the *ulReturn* field of the MCI_STATUS_PARMS data structure contains the current position of the device in MMTIME units.

(5) When the Caption Creation Utility program receives the position value, it writes the time value and the line of text to the caption file. The caption file contains the same text as the text file, but each line in the caption file is preceded by the time in the audio file when that line of text should be displayed.

CAPSAMP and CAPDLL

The Caption Sample Application (CAPSAMP) and Caption DLL (CAPDLL) are part of the sample captioning system provided with the Toolkit. See Captioning for additional information on this sample captioning system.

The Caption Sample Application and Caption DLL are provided to demonstrate how captioning can be integrated into applications using caption files in conjunction with the Caption DLL.

Program Flow

The following figure illustrates the interaction between the captioning components and the media control interface layer. Source code for the Caption Sample Application and Caption DLL are located respectively in the \TOOLKIT\SAMPLES\MM\CAPSAMP and \TOOLKIT\SAMPLES\MM\CAPDLL subdirectories.

```
       Caption
       Sample          (2D)
     Application

(2)(3)(4)(1)(2)(4)(5)


            Caption DLL   (2D)  Caption
                                 File

         (2B)
             (2C)
- -- -- -- --- -----------------------------------
                                  OS/2 Multimedia
    Media Control Interface
```

(1) As part of its initialization and termination routines, the Caption Sample Application issues ccInitialize and ccTerminate, respectively, to notify the Caption DLL to begin and end captioning. As part of the termination process, the Caption DLL releases any resources previously allocated for captioning.

(2) When you select **Play**, the Caption Sample Application opens the audio file, obtains a device ID, and plays the audio file. The Caption Sample Application queries the captioning flag. (You can set this flag by selecting the Captioning check box on the System page of the Multimedia Setup.) If this flag is set, the application issues ccSendCommand with a CC_START message to the Caption DLL. The Caption DLL then begins to provide captioning for the application.

(2B) When the Caption DLL receives the request to begin captioning, it issues a set position advise message to the multimedia system for every 1500 time units.

(2C) When the device moves 1500 time units, the Caption DLL receives an MM_MCIPOSITIONCHANGE message from the multimedia system.

(2D) Whenever the Caption DLL receives the MM_MCIPOSITIONCHANGE message, it checks the caption file for the appropriate line to display, based on the current position of the audio file. The Caption DLL then scrolls the caption window in the application, to display the appropriate line.

(3) If you pause the audio file, change the volume, or move the audio slider position, the Caption Sample Application does not have to perform any additional processing to manage the caption window. This processing is managed by the Caption DLL.

(4) When you select **Stop**, the Caption Sample Application sends an MCI_STOP message to the audio device. The application then issues ccSendCommand with a CC_STOP message, informing the Caption DLL to stop displaying the caption window in the application.

(5) You can change several properties of the caption window by selecting **Settings** from the Options menu of the Caption Sample Application. The application issues ccSendCommand with a CC_STATUS message to query the current properties of the caption window. When you select **OK** to save the desired properties, the Caption Sample Application issues the ccSendCommand with a CC_SET message to the Caption DLL. The Caption DLL handles changing and displaying the new properties of the caption window.

CLOCK - Memory Playlist Sample

The Memory Playlist Sample (CLOCK) illustrates the use of the memory playlist feature of OS/2 multimedia. The memory playlist feature provides for easy manipulation of multimedia data in memory to create unique effects based on user input or other dynamic events. In the case of the Memory Playlist Sample, multimedia data is manipulated dynamically to create sound effects (chimes) based on the time. The sample program also illustrates the use of the captioning flag provided by OS/2 multimedia. The Memory Playlist Sample uses this flag to decide if it should provide users with a visual cue of chimes.

Program Flow

The following figure illustrates the interaction between OS/2 multimedia system components and the Memory Playlist Sample. Source code is located in the \TOOLKIT\SAMPLES\MM\CLOCK subdirectory. (Source code for the Waveform Audio Media Driver is located in \TOOLKIT\SAMPLES\MM\ADMCT.)

```
 Memory Playlist    (2)   Audio
 Sample Program   `       File

   (1)

 Media Device
    Manager

   (3)

Waveform Audio
 Media Driver

   (3)

    Memory
   Playlist
```

(1) The Memory Playlist Sample queries the Media Device Manager using the mciQuerySysValue function to determine whether the captioning flag is set. If the flag is set, the bell in the primary window will swing back and forth when a chime occurs.

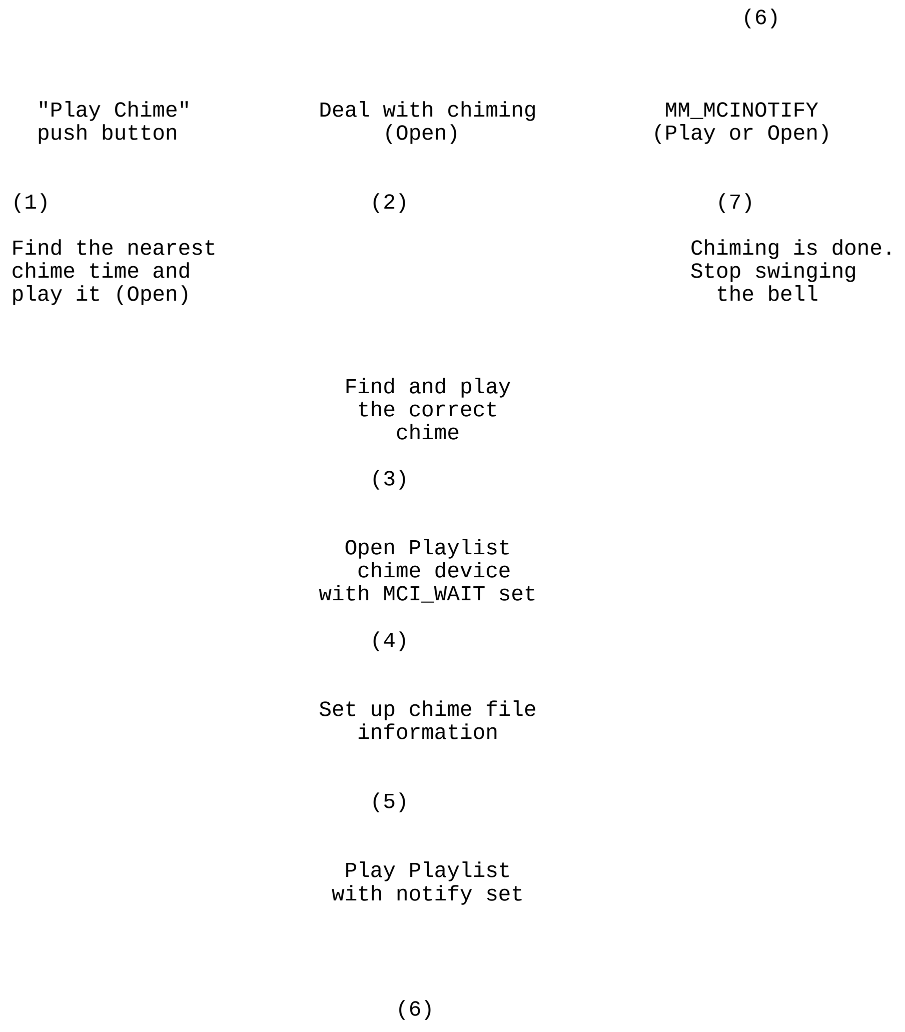
(2) The Memory Playlist Sample program initializes the playlist and loads the audio files into memory.

(3) When a chime occurs, the Memory Playlist Sample program calls the Waveform Audio Media Driver through the MDM to interpret instructions for the playing of three audio files in memory. The clock chimes are stored in three files: A, B, and C. Depending on the time a chime is requested, the memory playlist consists of the following:

- At 15 minutes past the hour, the chime plays A.
- At 30 minutes past the hour, the chime plays A+B.
- At 45 minutes past the hour, the chime plays A+B+A.
- On the hour, the chime plays A+B+A+B+ (C x Hour).

Playing a Chime

Every time you want to play a chime, open the device you want to play, play it, and close it. The logic for the Memory Playlist Sample implementation of the playlist is outlined in in the following figure.


(6)
"Play Chime" push button
Deal with chiming (Open)
MM_MCINOTIFY (Play or Open)
(1)
(2)
(7)
Find the nearest chime time and play it (Open)
Chiming is done. Stop swinging the bell
Find and play the correct chime
(3)
Open Playlist chime device with MCI_WAIT set
(4)
Set up chime file information
(5)
Play Playlist with notify set
(6)

(1) If **Play chime** is selected, processing is passed to the FindTheNearestChimeTimeAndPlayIt procedure, which calculates the next hour so its chime can be played. The procedure calls FindAndPlayTheCorrectChime to do the actual work required to play the chime.

(2) If it is a normally scheduled chime time (if a quarter hour has been reached), processing is passed to

FindAndPlayTheCorrectChime.

(3) FindAndPlayTheCorrectChime calls OpenPlaylistChimeDevice to do open processing with MCI_WAIT specified.

(4) When the OPEN command completes, the Waveform Audio Media Driver needs information about the waveform file. The FindAndPlayTheCorrectChime procedure calls SetupChimeFileInformation to pass this information to the driver.

(5) After calling SetupChimeFileInformation, the FindAndPlayTheCorrectChime procedure issues the mciSendCommand request to play the chime with the notify flag set.

(6) After the chime has finished playing, a notification message is sent to the MainDialogProc and handled by the MM_MCINOTIFY case of the switch statement.

(7) In the MM_MCINOTIFY case of MainDialogProc, when *usMCIUserParameter* is set to indicate the chime has stopped playing, the swinging of the bell is stopped and the device is closed.

DOUBPLAY - Double Buffering Playlist Sample

DOUBPLAY allows an application to play audio files directly from application memory buffers. An array of multiple playlist structures can be constructed that combines playlist commands with data buffers to perform complex operations on multiple buffers.

This sample takes a single wave file, MYWAVE.WAV, and plays it using a memory playlist. The playlist is constructed as a circular buffer composed of a series of small buffers, the sum of which may or may not be larger than the size of the .WAV file. This circular buffer is used to repeatedly play an audio file when the PLAY button is selected. As each buffer in the playlist is expended, the application refills the expended buffer with new data from the .WAV file. When all the buffers in the playlist have been expended, the playlist branches back to the first buffer to play the new data. This circular buffering process continues until the STOP button is selected or the application is closed.

DIVE - Direct Interface Video Extensions Sample

The DIVE Sample illustrates the use of direct interface video extensions (DIVE), which provides optimized blitting performance for motion video subsystems and applications that perform rapid screen updates in the OS/2 PM and full-screen environments. Using DIVE interfaces, applications can either write directly to video memory or use the DIVE blitter. The DIVE blitter will take advantage of acceleration hardware when present and applicable to the function being performed.

Using DIVE, the sample program blits a series of 16 compressed 256-color bit maps to the screen. For more information on how to use the DIVE functions included in this sample program, see Direct Interface Video Extensions (DIVE).

Note: The DIVE sample requires OS/2 Warp, Version 3 in order to execute properly. The files for the samples will be installed when the samples are selected, but Workplace Shell objects will not be created for them if the installed operating system is not OS/2 Warp, Version 3.

The OS/2 Warp color support defaults to 16 colors. This means that your setup needs to be updated, otherwise the DIVE sample will not run.

The maximum window size of this sample has been limited to 640x480 because larger window sizes may cause excessive swapping on machines with less than 16MB.

Duet Samples

This section describes the use of the media control interface layer of the OS/2 multimedia system to manipulate devices in a hardware-independent manner. The Duet sample programs (DUET1 and DUET2) illustrate the OS/2 multimedia concept of device grouping and integrating multimedia into an application's help information. See the example in the Duet Player IPF Sample section for an example of how multimedia can be used in an application's help information.

The Duet sample programs communicate with the media control interface subsystem to control multimedia devices from an application in a device-independent manner. This concept is illustrated by the Duet samples when they play, pause, resume, stop, and change the volume of their songs. This function is implemented using the procedural interface, as opposed to the string interface described in the MCISTRNG - Media Control Interface String Test Sample.

The Duet sample applications are identical except for minor software differences in how the hardware devices are controlled. DUET1 plays both parts of the duet on a *streaming device* -a device that streams data through the SPI subsystem of OS/2 multimedia. DUET2 groups a streaming and a *non-streaming device* -a device that handles data internally and does not stream data through the SPI subsystem.

The hardware differences between the Duet samples have little impact on the applications. The media drivers and stream handlers manage the hardware-dependent function.

Note: To hear the duets when you run the Duet sample programs, an audio device is required, for example, the M-Audio adapter. If you do not have an audio adapter, you can still compile and run the sample programs; however, you will not be able to play any duets or audio help.

DUET1 - Streaming Device Duet Sample

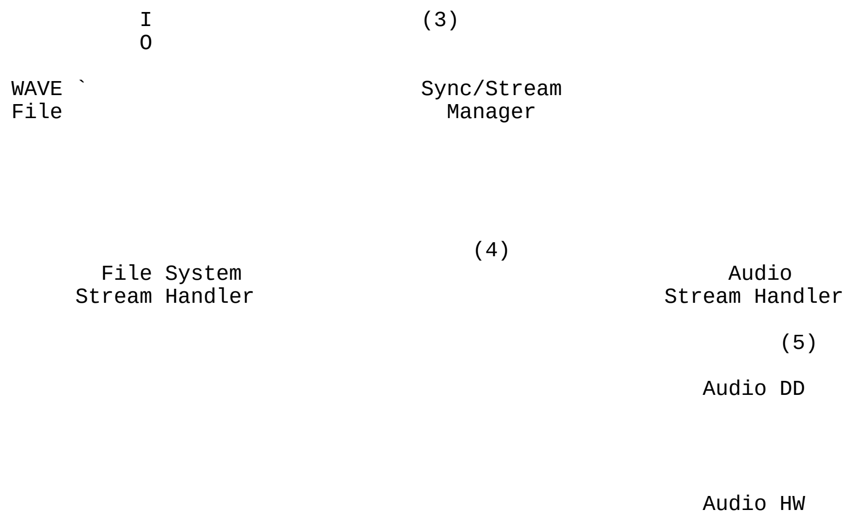
The Streaming Device Duet Sample (DUET1) illustrates the concepts of grouping two streaming devices. Each song that is played by DUET1 is recorded into two separate waveform files (instead of a single waveform file, which is the most common and efficient means to record). Think of each song as a duet, with each part of the duet stored in a separate waveform file.

Note: The DUET1 program will not execute correctly when using a Sound Blaster adapter because the adapter does not support playing two wave files simultaneously.

Program Flow

The following figure illustrates how the DUET1 sample program interfaces with the Media Device Manager (MDM) to handle the concept of grouping. Source code for DUET1 is located in the \TOOLKIT\SAMPLES\MM\DUET1 subdirectory.

```
                                   DUET1
                                   Sample
                                   Program

                                     (1)


                    Waveform                    Waveform
                      Audio                       Audio
                 Device Context 1    (2)     Device Context 2



                                Media Device
                                   Manager




 WAVE
 File `                           Waveform
                                Audio Media
           M                       Driver
           M
```



(1) The DUET1 program directly interfaces with media control interface services using the procedural interface.

Each part of the duet is played through the OS/2 multimedia system using a separate instance of a logical media device. In this case, it uses two instances of the logical waveform device, the Waveform Audio Media Driver. When the Duet Player is ready to play a song, it issues the MCI_GROUP command to the MDM to set up a group of two logical waveform devices. The MDM returns a group handle to the application, so that the application can refer to that group as a single unit (instead of having to manage each logical device separately). The MDM handles the management of the logical media devices for the application.

(2) The MDM sets up two Waveform Audio Media Drivers as a group.

(3) The Waveform Audio Media Driver handles the creation and management of the source and target stream handlers. In this example, the source stream handler is the File System Stream Handler, and the target stream handler is the Audio Stream Handler. When the Waveform Audio Media Driver receives an MCI_OPEN command from the program, it creates the stream handlers.

When you select **Play**, **Pause**, or **Stop**, the application issues an MCI_PLAY, MCI_PAUSE, or MCI_STOP message to the MDM. The MDM then passes this message to the appropriate media devices in the group.

(4) The media driver issues SPI functions to the Sync/Stream Manager (SSM) to process the play, pause, resume, and close messages from the application. The SSM controls the registration and activities of all stream handlers. SPI services enable media drivers to create, start, and end data streams. SPI also provides real-time services to enable stream handlers to synchronize events. The File System Stream Handler and the Audio Stream Handler are responsible for controlling the flow of application data in a continuous, real-time manner.

(5) The target stream handlers interface with the audio device driver through the inter-device-driver communication (IDC) interface. This is the interface between the stream handler and the physical device driver (PDD). The IDC interface includes Stream Handler Device (SHD) messages and device driver command (DDCMD) messages.

DUET2 - Streaming and Non-Streaming Device Duet Sample

Code for the Streaming and Non-Streaming Device Duet Sample (DUET2) is almost identical to DUET1. The difference is that DUET2 demonstrates how one of the devices in the multimedia device group can be a non-streaming device. The song in DUET2 is divided in two parts, just as the song in DUET1. However, DUET2 plays one part of the song from a CD in a CD-ROM device connected to the system. This difference in hardware is almost totally isolated from the application. There is a difference of approximately 10 lines of code between the Duet sample programs to handle the major difference in hardware configurations.

This major difference in hardware configuration is buffered from DUET2 by the media drivers. The CD Media Driver performs differently from the Waveform Audio Media Driver because it is dealing with a non-streaming device. When the CD media driver receives a media control interface message from the program (such as PLAY or STOP), it must communicate with the CD-ROM device driver through the IOCtl interface. This is the difference between a media driver that controls a streaming device and a media driver that controls a non-streaming device.

MCISPY - MCISpy Sample

The MCISpy Sample (MCISPY) monitors media control interface messages that are exchanged between applications and the OS/2 multimedia subsystem. In addition to teaching you about multimedia messages, MCISpy also serves as a debugging aid.

To begin monitoring messages and saving them in a log file:

1. Start MCISpy.
2. Select **LogFile** from the MCISpy menu bar.
3. Select **Enable Message Logging**.
4. Start the multimedia application you want to monitor.

As you interact with the application, information is displayed in the MCISpy window and written to the default log file MCISPY.LOG. MCISpy monitors both the string interface and the command message interface.

For the command message interface, the information consists of:

- Originating process ID (PID)
- Device ID
- Message name
- Message flag value
- User parameter
- Data

For the string interface, the information consists of the PID and the string input.

To specify a log file other than the default log file, select **Open**. Use the **Edit** functions to manage the contents of the log file.

The **View** menu provides views of useful information. The Installed Devices view lists the logical and physical device names and product information for each installed multimedia device. The MCI Flag Values view shows you the text string equivalents for the hexadecimal flag values that accompany the messages displayed by MCISpy during the monitoring process.

The **Filter** menu offers a variety of filters for the messages that are displayed in the log file. You can filter the messages by:

- Selecting the message names you want to see.
- Selecting the device type or open device ID of the device whose messages you want to see.
- Disabling input from mciSendCommand, mciSendString or mdmDriverNotify functions. This choice takes precedence over any other filters in effect.

Initially no filters are in effect and therefore all messages are displayed.

Program Flow

The following diagram depicts the MCISpy application implementation model. The stub or proxy DLL replaces the true MDM.DLL, which is renamed to MCI.DLL. The proxy MDM exports the same APIs at the same ordinals as the true MDM. The proxy MDM has forwarder entries into the MCI.DLL for the API workers. That is, the proxy exports and also imports the same APIs from the true MDM.

```
                   Shared Memory
                    (Named)

Exported APIs                                             True MDM
                   Stub or                                containing
mciSendCommand()   Proxy            Forwarder entry       the API
                   MDM.DLL                           >    workers is
mciSendString()                     Forwarder entry       renamed to
                   Maintain Open                     >    MCI.DLL.
```

```
mdmDriverNotify()  Device Ids       Forwarder entry
                                                       >
                   Notify Spy
                   Application


                   Shared Memory
                    (Named)

                   MCISpy
                   Sample Program
```

MCISTRNG - Media Control Interface String Test Sample

The String Test Sample serves as a powerful testing and debugging tool that enables developers writing media drivers to control their devices at the application level. The String Test Sample illustrates how an application uses the interpretive string interface provided by the media control interface. It also illustrates how notification messages are returned from the media drivers to the application.

The media control interface of OS/2 multimedia provides the primary mechanism for application control of media devices. Applications interface with the media control interface (and thus with media devices) in two ways-through a procedural interface or through a string interface. The procedural interface (also referred to as command message interface) is used for sending messages to the media control interface from an application using the mciSendCommand function. The string interface is used for sending command language statements to the media control interface from an application using the mciSendString function.

The string interface has a few advantages over the procedural interface. First, the string interface enables you to interactively control devices with a PM or command line interface. In addition, applications that currently use script languages can integrate "string commands" in their script languages allowing you to integrate multimedia in your applications with a very low cost and development impact. Finally, in certain programming circumstances, the string interface is also easier to use than the procedural interface. The string interface simply passes a character-string buffer to the function, in contrast to the procedural interface, which requires the setup of certain data structures.

Note: The procedural interface is illustrated in the Duet Samples.

Program Flow

The following figure illustrates the interaction between OS/2 multimedia system components and the Media Control Interface String Test Sample program. Source code is located in the \TOOLKIT\SAMPLES\MM\MCISTRNG subdirectory.

```
  String Test
 Sample Program


(4)    (1)

  Media Device  (2)     Command
     Manager     `       Table


(4)    (3)

  Media Driver
```

(1) String Test waits for you to enter a string command. When you select **Send**, the program calls mciSendString to send the

string command to the Media Device Manager (MDM).

(2) The MDM receives the string command and uses the command table to parse the string and convert it to the media control interface command format.

(3) The Media Device Manager passes the command to the appropriate media driver, which acts on the command.

(4) Notification messages are sent back to the application from the media device to indicate such operations as completion of a device function or the passing of device ownership from one process to another. These messages can be viewed in the Display Messages dialog box of the String Test Sample.

MMBROWSE - Image Browser Sample

The Image Browser Sample (MMBROWSE) illustrates how to use the multimedia I/O subsystem (MMIO) to install I/O procedures for various image formats and then convert these image formats to any of the supported image formats.

This sample demonstrates the various ways to install an I/O procedure:

- On a temporary basis for an individual file.
- On a semi-permanent basis for the duration of a process.
- On a permanent basis for use by any process in the system.

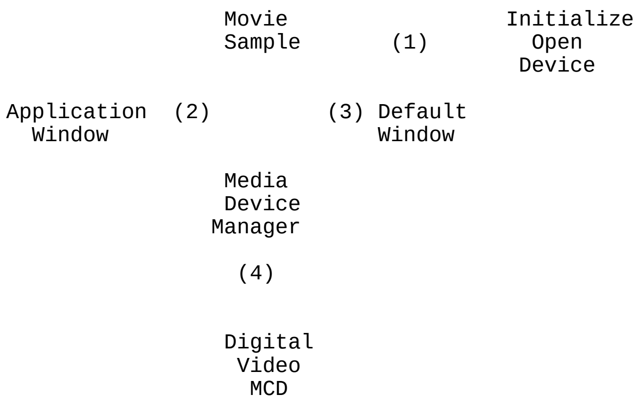
To install an I/O procedure, you must know its four-character code. For more information on how to use the functions included in this sample program, see Multimedia I/O File Services.

MOVIE - Movie Sample

The Movie Sample (MOVIE) demonstrates device control of a software motion video device. It also illustrates how to cut, copy, paste, and delete movie data from an application. A movie can be played in an application-defined window or in the system default window provided by the software motion video subsystem. The Movie Sample program uses the string interface to interact with the media control interface layer.

Program Flow

The following figure illustrates the interaction between the Movie Sample program and the Media Device Manager (MDM). Source code for the Movie Sample is located in the \TOOLKIT\SAMPLES\MM\MOVIE subdirectory.



(1) The Movie Sample creates a standard window with a style of CS_MOVENOTIFY for the application window during initialization. The Movie Sample also initializes and loads a movie file into memory during initialization time.

(2) If you choose to play the movie in an application window, three string commands are generated and sent to the digital video MCD. They are **window**, **load**, and **put**. The following list describes these commands.

window Specifies the characteristics of the application-defined window.

Note: When a window handle is selected with the **window** command, the digital video MCD queries the size and position of the window. If the window is visible, the digital video MCD alters the size and position of the video accordingly. However, if the window size and position is changed while the window is hidden, the digital video MCD does not note the new size and position.

load Loads the digital video file.

put Places the movie in the application-defined window.

These commands (**window**, **load**, and **put**) must be issued in sequence to play the movie in the application-defined window.

(3) If you choose to play the movie in a default window, the **load** command is generated and issued to the digital video MCD.

(4) When you select **Play**, the Movie Sample program calls the digital video MCD, through the Media Device Manager, to play the movie file.

If you select **Edit**, processing is passed to the EditTheMovie procedure which loads the movie, and sets the data format to frames. It copies the first 25 frames to the clipboard. The 25 frames are pasted four times to the end of the movie file (duplicating the first 25 frames four times). Next, the EditTheMovie procedure cuts the first 10 frames from the movie, seeks to the end of the movie, and pastes the 10 frames to the end. The **cut** command deletes the specified number of frames from the position the application specifies to start the cut. Once the movie is edited, selecting **Play** causes the edited version of the movie to play. To play the original "unedited" movie file, exit the Movie Sample program, restart it, and select **Play**.

RECORDER - Audio Recorder Sample

The Audio Recorder Sample (RECORDER) illustrates the concept of recording audio through the media control interface and how to query a device to find out the recording capabilities. The sample program also illustrates how to change the audio recording and audio device properties, such as bits per sample, samples per second, input level, and input source.

Program Flow

The following figure illustrates how the Audio Recorder Sample program interfaces with the Media Device Manager (MDM). Source code for the Audio Recorder Sample is located in the \TOOLKIT\SAMPLES\MM\RECORDER subdirectory.

```
        Audio Recorder
           Sample

        (1)      (2)


       Media Device
        Manager

    (1) (2)      (2)
```

```
 Wave Audio   Amp-Mixer
    MCD          MCD
```

(1) Upon initialization, the Audio Recorder Sample program issues an MCI_GETDEVCAPS command, with the MCI_GETDEVCAPS_EXTENDED flag set. The Audio Recorder Sample uses the device capability information that is returned to disable any options in the Audio Recorder Sample menus that do not apply to the installed audio device.

(2) When you select **Record**, the Audio Recorder Sample program checks all of the menu settings and sets up the device accordingly. The Audio Recorder Sample issues an MCI_SET command to the Waveform Audio Media Driver to set up the bits per sample, samples per second, and channels of the recording. The sample also issues an MCI_CONNECTOR command to the Waveform Audio Media Driver to set up the input source. It gets the device ID of the Amp-Mixer Media Driver and issues an MCI_SET command to that device to set up monitor input and input level. It also sends an MCI_SET command to the Amp-Mixer Media Driver to set the input level of the recording. Finally, the Audio Recorder Sample sends an MCI_RECORD command to the waveaudio device to begin recording.

TUNER - TV Tuner Sample

TV tuner cards allow a desktop PC to receive and display broadcasted television signals. TUNER provides an example of how to use the MCI string interface to control a tuner card.

Note: A TV tuner card (such as the WinTV Basic) is required to run the TUNER sample.

Once the application is running, the display window can be sized and a TV channel can be selected. The channel can be selected through up/down buttons or a text entry field displayed both at the bottom of the window.

A TV tuner card is handled as a digital video device by the multimedia subsystem. A typical system has one or more digital video devices, with digitalvideo01 assigned to software motion video. By default this sample opens the digitalvideo02 device. If the tuner card is not assigned to digitalvideo02 an alternate device ordinal must be provided at the command line with the format:

```
  /d=digitalvideoxx
```

where *xx* is a two digit number, padded on the left with zero.

For example:

```
  /d=digitalvideo03
```

Most TV tuner cards, such as the Hauppauge WinTV card, support several video sources. The connector number that corresponds to the tuner video source is dependent on the hardware and may not be the same for each tuner card. The connector number is not changed by this sample. Instead, it uses the default connector number set in the Multimedia Setup application.

ULTIEYES - Non-Linear Video Sample

The Non-Linear Video Sample (ULITEYES) demonstrates the use of non-linear video by displaying segments from a movie clip in response to input from the mouse. Whenever the mouse changes position, ULTIEYES calculates the angle of the mouse pointer to the center of the ULTIEYES video window. It then displays a frame from the movie ULTIEYES.AVI that corresponds to the position of the mouse pointer. As the mouse pointer is moved around the desk top, the eyes shown in the video window appear to follow the mouse. The movie file also contains segments of the eyes winking and blinking. ULTIEYES shows these segments when it detects a mouse button 1 click on the left side, center or right side of the video window.

Creating a Movie Clip

If you have Video IN and one of the supported video capture cards, you can easily replace ULTIEYES.AVI with a movie clip of your own eyes. Make the following updates in the Video IN Recorder Settings notebook before you record the movie:

1. Set the reference frame rate to **1**.
2. Set the audio to **Record no audio**.

After you record the movie, you are ready to select the frames the ULTIEYES program will display when the mouse pointer is at various positions in relation to ULTIEYES: East, East by Northeast, Northeast, and so on.

1. Set the time format of the Video IN Recorder to frames.
2. Step through the movie frame by frame.
3. Jot down the frame number for each position on the compass. Also note the beginning and ending frames of the segments for the left and right winks and the blink. (Note that video segments must be at least one second in length.)
4. Create an ASCII file that contains 25 lines in the following format:
    - The first line in the file is the name of the AVI movie file.

    - The next 24 lines comprise the frame number table (0-23). A description of the table contents is shown in the following figure. Letter characters on each line indicate the direction of the gaze associated with that particular line number. For example, in the frame on line 12 the eyes should be gazing south by southwest. To see what an actual frame table looks like, open the ULTIEYES.EYE file.

```
 0      - Eyes turned inward frame.
 1      - E
 2      - ENE
 3      - NE
 4      - NNE
 5      - N
 6      - NNW
 7      - NW
 8      - WNW
 9      - W
10      - WSW
11      - SW
12      - SSW
13      - S
14      - SSE
15      - SE
16      - ESE
17      - E (again)
18-19   - Right-eye wink sequence (left side of video)
20-21   - Left-eye wink sequence (right side of video)
22-23   - Blink sequence
```

To start your version of ULTIEYES, enter:

```
ULTIEYES filename
```

where *filename* is the name of the ASCII file you created.

Installing a Program Using MINSTALL

This section describes the procedures for preparing control files for installing a multimedia application using the multimedia installation program (MINSTALL).

The MINSTALL program utilizes features provided by the OS/2 multimedia installation engine, such as creating workplace shell objects and changing the CONFIG.SYS file, providing a consistent installation process.

Note: For installation requirements when writing a subsystem, refer to the *OS/2 Multimedia Subsystem Programming Guide* .

Installation Overview

The MINSTALL program (MINSTALL.EXE) requires specific file information to install each program. This file information is provided by the master control file CONTROL.SCR.

The master control file, CONTROL.SCR, tells the installation program what to install, where to install it, how to display it to the user, and what system files need to be updated. CONTROL.SCR uses keywords to specify these instructions to MINSTALL. (See Master Control File.)

The following important keywords used in CONTROL.SCR specify subsequent control files that list the installation files, create necessary entries in the initialization file, and add appropriate lines in the CONFIG.SYS file:

- The keyword FILELIST specifies the name of the file list control file, which lists all the files in the installation package. (See File List Control File.)

- The keyword SSINICH identifies a control file that is used to define folders and programs.

- The keyword SSCONFIGCH identifies a control file that contains all the changes to be made to the CONFIG.SYS file for a particular installable item or subsystem.

For each installation, an error log called MINSTALL.LOG is created. If a problem occurs during installation, the problem is recorded in the error log and might also be displayed on the screen. You can read the log for more information about the error.

Master Control File

The master control file, CONTROL.SCR, contains information about each installable feature. It identifies specific file information that MINSTALL requires to install the requested features. This control file must be named CONTROL.SCR and must reside on the first media unit (diskette or CD) of the installation package.

During installation, if any errors are detected in the master control file, the main installation selection window will be empty and the errors will be logged in the error log MINSTALL.LOG. You can access this file for information about the errors.

The CONTROL.SCR file consists of a header section and a subsystem definition section. The header section comes first and provides general information for the installation procedure. The subsystem definition section comes next and provides information about the features in the installation package.

CONTROL.SCR Header

The header section of the CONTROL.SCR file contains the following information:

- Name of the installation package
- Code page used when creating the file
- Name of the file list control file
- Number of features in the installation package
- Number of media units required for installation
- Names of the media units required for installation
- Source and destination directory names (optional)

The following is an example of the CONTROL.SCR header located in the \TOOLKIT\SAMPLES\MM\SHORTCF subdirectory.

```
package="IBM Multimedia Presentation Manager Toolkit/2"
codepage=437
filelist="filelist.tk2"
groupcount=2
munitcount=1
medianame="IBM Multimedia Presentation Manager Toolkit/2 Installation
          Diskette 1"

sourcedir = "\\"                                   = 4
sourcedir = "\\MCISTRNG\\"                         = 11
```

```
destindir = "\\MMOS2\\"                                = 0
destindir = "\\MMOS2\\INSTALL\\"                       = 4
destindir = "\\MMOS2\\MMTOOLKT\\SAMPLES\\MCISTRNG\\"   = 11
```

A larger and more complex example of installing multiple items using MINSTALL is located in the \TOOLKIT\SAMPLES\MM\CF subdirectory.

The following table describes the keywords used in the CONTROL.SCR header.

Keyword	Description
PACKAGE	This required keyword specifies the name of the installation package in the form of a quoted string. For example: PACKAGE="IBM Multimedia Presentation Manager Toolkit/2"
CODEPAGE	This required keyword specifies the code page that the file was created under. For example: CODEPAGE = 437
FILELIST	This required keyword specifies the name of the file list control file. This control file contains a list of files that make up each subsystem, identifies on which media units they reside in the installation package, and specifies the destination to which they will be copied. For example: FILELIST = "FILELIST.TK2"
GROUPCOUNT	This required keyword specifies the number of subsystems in the installation package. All groups are counted, including group 0 (if present). For example: GROUPCOUNT = 2
MUNITCOUNT	This required keyword specifies the number of media units (diskettes, CDs) that will be used if all features are installed. This number must be greater than 0. This is the number of diskettes or CDs on which the installation package resides. For example: MUNITCOUNT = 1
MEDIANAME	This required keyword specifies a unique media name, which is a character string on the diskette or CD label. For each media unit, this keyword must be repeated once, in the form of a quoted string. This information is used during installation to prompt the user to insert a diskette or CD when needed. For example: MEDIANAME = "IBM Multimedia Presentation Manager Toolkit/2 Installation Diskette 1"
SOURCEDIR	This optional keyword specifies the name of a source directory and its associated number. This keyword can be repeated and is specified by a quoted string followed by an equal sign (=) and a number. The number is used to identify the particular directory in later scripts. This can be NULL, in which case two default backslash characters (\\) are used with an encoding of 0. The path must be surrounded by path separators. For example: SOURCEDIR="\\LIB\\" = 1
DESTINDIR	This optional keyword specifies the destination directory and its encodings. This keyword can be repeated and can be NULL, in which case the default is the

```
                  \MMOS2 subdirectory.  The path must be
                  surrounded by path separators and any
                  directory that does not exist will be
                  created. For example:
                  DESTINDIR = "\\MMOS2\\" = 0
```

Observe the following guidelines when you create or change a CONTROL.SCR header:

- You must place the keywords MUNITCOUNT and MEDIANAME so that MUNITCOUNT comes *directly* before MEDIANAME. The order of the other keywords is not significant.
- The destination directory (DESTINDIR) must have a unique number.
- A subsystem group can be spread across several media units. It does not have to reside on one media unit.
- A subsystem can define any directory. MINSTALL will create any subdirectories defined with the DESTINDIR keyword that do not exist.
- If you move the installation package files to media of a different size, the number of media units (MEDIACOUNT) might change.
- You may use comments in the header section in the form of blank lines or text enclosed with "/* and */". You may not use nested comments.
- You may use blank spaces around the equal sign; blank spaces are ignored.
- If you want to use a double quotation mark or a backslash in a string, you must precede it with the escape character (\).
- You may use the escape sequence \n (new line).

CONTROL.SCR Subsystem Definition

The subsystem definition section of the CONTROL.SCR file follows the header section and contains the definitions for all the installable items in the installation package. A block of information must be included for each subsystem.

The subsystem definition section contains the following information:

- The group or item number
- The item name
- The version of the item
- The size of all the files for the item installation
- The names of the control files that change the MMPM2.INI and CONFIG.SYS files
- The names of installation DLL files and entry points

The following is an example of a CONTROL.SCR subsystem definition.

```
ssgroup=0              /*  base group */
sssize=41
ssname="mmbase"
ssversion="1.0.9"
ssinich="TLKBASE.SCR"

/*           - 11 = mcistrng */
ssgroup=11
ssname="Media Control Interface String Test"
ssversion="1.0.9"
sssize=200
ssinich="TLKSTRN.SCR"
ssicon="mcistrng.ico"
```

The CONTROL.SCR subsystem definition consists of the following keywords.

```
 Keyword               Description
```

SSGROUP

This required keyword specifies the group or item number. This marks the beginning of a group for this item and assigns it a number. Each item must have a unique number from 0-49 within the package; however, the same number can be used with different installation packages. The groups are displayed in the installation main selection window in ascending order by group number. For example:
SSGROUP = 11

SSNAME

This required keyword specifies the item name, which names the current group as an ASCII string. This keyword is case sensitive and takes the form of a quoted string. The name may include special characters and may be translated. The name is displayed in the main installation selection window. For example:
SSNAME = "CD Audio"

SSVERSION

This required keyword specifies the version of the component in the form of a quoted string. This string must be in the format "*dd.dd.dd*" (where *dd* represents digits). Any version not specified in this format will be converted to that format. All string items that are not digits or periods will be converted to zeros. Any periods after the second period will be converted to zeros. For example:
SSVERSION = "1.1.0"

SSSIZE

This required keyword specifies the total size of all the files in the item. The size denotes the number of bytes in thousands (500 = 500KB). This number is used to help determine if there is enough disk space to support the installation. If you do not know the correct size of a item, overstate its size. For example:
SSSIZE = 1024

SSINICH

This optional keyword specifies the name of the file that contains the changes to the MMPM2.INI file. If this statement is missing, there are no changes to the MMPM2.INI file. For example:
SSINICH = "ACPAINI.CH"

SSCONFIGCH

This optional keyword specifies the name of the file that contains the changes to the CONFIG.SYS file. If this statement is missing, there are no changes to the CONFIG.SYS file. For example:
SSCONFIGCH = "ACPACON.CH"

SSCOREQS

This optional keyword specifies a list of corequisites needed for this item to run. It also specifies what other components the current components depend on. These other components must be installed for the current component to function. (If this statement is missing, there are no corequisites.)
The corequisite is identified by its group number. Corequisite groups should point to each other only if they require each other. It is possible to have subsystem A list subsystem B as a corequisite and subsystem B have no corequisites. If the user selects a subsystem with a corequisite, but does not select all corequisites, the user is

notified before the installation starts. This entry can be repeated as necessary. For example:
SSCOREQS = 1

SSICON

This optional keyword names the icon file for this component that is to be displayed in the main installation selection window. The icon file to be displayed in the selection window must reside on the first installation media unit. If this statement is missing, a default icon is used. For example:
SSICON = "ACPA.ICO"

SSDLL

This optional keyword names a DLL file that is to be run during the installation process. The DLL referenced will be run after all files are copied to the destination, but before any script processing is performed. If this keyword is present, the SSDLLENTRY keyword must also be present. For example:
SSDLL="MY.DLL"

SSDLLENTRY

This optional keyword specifies the name of the entry point into SSDLL in the form of a quoted string. If this keyword is present, the SSDLL keyword must also be present. For example:
SSDLLENTRY="MyEntry"

SSTERMDLL

This optional keyword names a DLL file that is to be run during the installation process. The DLL referenced will be run after all files are copied to the destination and after all script processing is done. The purpose of this keyword is to allow for processing to occur on a fully configured multimedia system. If this keyword is present, the SSTERMDLLENTRY keyword must also be present. For example:
SSTERMDLL="MYTERM.DLL"

SSTERMDLLENTRY

This optional keyword specifies the name of the entry point into SSTERMDLL in the form of a quoted string. If this keyword is present, the SSTERMDLL keyword must also be present. For example:
SSTERMDLLENTRY="MyTermEntry"

SSSELECT

This optional keyword determines the preselection of items for installation. Five values are supported:
"ALWAYS" - This value specifies that the group will always be installed. It is the only valid value for Group 0. Groups with this value will not be displayed in the installation selection window. This is the default.
"REQUIRED" - This value specifies that the group will be preselected for installation. If the group had been previously installed, it cannot be unselected by the user if this installation package is newer than the installed version. If the group has not been previously installed, it can be unselected by the user.
"VERSION" - This value specifies that the group will be preselected only if it was previously installed and this installation package is newer than the installed version. However, it can be unselected by the user.

```
"YES" - This value specifies that the
group will be preselected whether or not
it was previously installed.  It can be
unselected by the user.
"NO" - This value specifies that the
group is never preselected but can be
selected by the user.
"BASENEWER" - This value specifies that
files belonging to this group will only
be copied if the user's machine has no
package installed or if the package
installed is older than the current
package.
"ONLYNEWER" - This value specifies that
a user will not be able to to install an
older version of a package on top of a
newer version. Files belonging to this
group will only be copied if the user
has an older version (or the same
version) installed.  If no version is
installed or if the version installed is
higher than the one in the package, no
files will be copied.
```

Observe the following guidelines when you create or change a CONTROL.SCR subsystem definition:

- The SSGROUP keyword must be the first statement in the information block.
- An item may reside on different media.
- Each statement in the information block must have a value.
- You may use comments in the header section in the form of blank lines or text enclosed with "/* and */". You may not use nested comments.
- You may use blank spaces around the equal sign; blank spaces are ignored.
- If you want to use a double quotation mark or a backslash in a string, you must precede it with the escape character (\).
- You may use the escape sequence \n (new line).

File List Control File

The master control file, CONTROL.SCR, specifies a FILELIST keyword which identifies the name of a file list control file that lists all the installable files in the installation package. The file list control file also contains the following additional information:

- The name of the file
- The number of the media unit where the file is stored
- The destination directory where the file will be copied
- The group or feature the file is identified with

The following is an example of the file list control file (FILELIST.TK2) located in the \TOOLKIT\SAMPLES\MM\SHORTCF subdirectory. The first nonblank, noncomment line is a count of the number of files (or file name lines) in the file.

```
/***********************************************************************/
/* This file contains install information.  Comments are delimited     */
/* as these comments are.  Blank lines are ignored.  Non-blank lines   */
/* will be parsed and extraneous characters will cause errors.  First  */
/* non-comment line must be the total number of files to be installed. */
/***********************************************************************/
/* all files on the install disk(s) are listed below. Other            */
/* information is also given, as follows:                              */
/*                                                                     */
/* Disk#    - The number of the disk on which the file resides.        */
/*            (Ignored if installing from CD-ROM). These are sorted    */
/*            from 0 to the number of disks, ascending.                */
/*                                                                     */
/* Group#   - The logical group to which the file belongs. Group       */
/*            starts at 0.                                             */
/*                                                                     */
/* Dest#    - The destination subdirectory into which the file will be */
/*            copied.  Dest# starts at 0.                              */
/*                                                                     */
/* Source #- The installation disk(s) subdirectory where the file      */
/*            resides.                                                 */
/*                                                                     */
/* FileName - The base filename.                                       */
/*                                                                     */
/*sourcedir="\\"                     = 4                               */
/*sourcedir="\\MCISTRNG\\"           = 11                              */
/*                                                                     */
/* destindir="\\MMOS2\\"                                 = 0           */
/* destindir="\\MMOS2\\INSTALL\\"                        = 4           */
/* destindir="\\MMOS2\\MMTOOLKT\\SAMPLES\\MCISTRNG\\"   = 11           */
/*                                                                     */
/*        groups                                                       */
/*                0 = Base                                             */
/*            - 11 = MCI String Test                                   */
/*                                                                     */
/**********************************************************************/
/* Total number of entries is 20. This must be the first parameter other*/
/*than comments*/
/**********************************************************************/

    20

/*It is a good practice to make groups and list the amount of files in
/*them (9 files)*/
/*A comment can not be the last line in a filelist.*/

/*            mmtoolkt\samples\cf  9   41K           */

    0    0   11     4    "CONTROL.SCR"
    0    0   11     4    "FILELIST.TK2"
    0    0   11      4   "TLKSTRN.SCR"
    0    0   11     4    "TLKBASE.SCR"
    0    0    4     4    "TLKSTRN.SCR"
    0    0    4     4    "TLKBASE.SCR"
    0    0    4     4    "MCISTRNG.ICO"
    0    0    4    4     "MMTOOLKT.ICO"
    0    0   11    4    "TOOLKIT.CH"

/* 11 files */
/*           mmtoolkt\samples\mcistrng  11   197K   */
    0   11   11   11   "mcistrng.c"
    0   11   11   11   "mcistrng.h"
    0   11   11   11   "mcistrng.rc"
    0   11   11   11   "mcistrng.dlg"
    0   11   11   11   "mcistrng.def"
    0   11   11   11   "makefile"
    0   11   11   11   "UMB.DAT"
    0   11   11   11   "mcistrng.exe"
    0   11   11   11   "mcistrng.hlp"
    0   11   11   11   "mcistrng.ipf"
    0   11   11   11   "mcistrng.ico"
```

The following table describes the columns in the file list control file.

Column	Description
Media#	Specifies the number of the media unit (diskette or CD) where the file is stored. The units are numbered starting from 0. This number will be used for all installation media except for the hard disk. The Media# column must be sorted in ascending order. A media unit does not have to be filled (there can be unused space on any numbered unit).
Group or subsystem#	Specifies the group to which the file belongs. The group or item number must be a positive integer, with numbering starting at 0 (the groups are defined in CONTROL.SCR by the SSGROUP keyword). This number is used to determine which files belong to a item selected for installation.
Destination#	Specifies the destination subdirectory where the file will be copied. The destination number is defined in the CONTROL.SCR file by the DESTINDIR

```
                          keyword.  This field must always be a defined
                          number (for example, 14 for the \MMOS2\DLL path).
                          If you specify a DESTINDIR statement in the master
                          control file, you only have to specify the
                          corresponding group number (for example, 1).

 Source#                  Specifies the path name of the source file.  The
                          source number is defined in the CONTROL.SCR file
                          by the SOURCEDIR keyword.  This field must always
                          be defined with a number (for example, 1 for the
                          \LIB path).

 File name                Specifies the base file name, which must be in
                          double quotes.  For example, "MINSTALL.EXE".
```

Change Control Files

Change control files are script files that make changes to the CONFIG.SYS file and INI files. The master control file, CONTROL.SCR, identifies the change control files when you specify the SSINICH and SSCONFIGCH keywords.

Supported Macros

Macros can be used in the change control files. Macros can also be used in the master control file. When a supported macro is used, drives and paths do not have to be identified until the time of installation. At installation, macros can perform the following:

- Replace the full destination path of the file
- Replace the installation target drive letter
- Replace the default destination drive and path as defined in CONTROL.SCR
- Replace the startup (boot) drive letter of the operating system
- Delete specified files

The following table describes the supported macros. Refer to the TLKSTRN.SCR file in the \TOOLKIT\SAMPLES\MM\SHORTCF subdirectory for an example.

Macro	**Description**
destindir= "$(DELETE)\\path\\" = *number*	The $(DELETE) macro can only be used with the DESTINDIR keyword in the master control file. The relative *number* is listed in the file list control file. Every file that has this number will be deleted from the user's machine.

$(DEST)*filename*	$(DEST) is replaced with the full destination path of the file. For example:
	`$(DEST)CLOCK.EXE`
	becomes
	`D:\MMOS2\MMTOOLKT\SAMPLES\CLOCK\CLOCK.EXE`

$(DRIVE)	$(DRIVE) is replaced by the installation target drive letter. (Do not append a colon.) For example: `$(DRIVE)\MMOS2\TEST1.EXE`

becomes

```
D:\MMOS2\TEST1.EXE
```

$(DIR)#

is the number of the destination directory as stated in the file CONTROL.SCR. The macro is replaced by the default drive and path defined in the CONTROL.SCR file for the specified DESTINDIR definition. For example:

```
$(DIR)4\MINSTALL.LOG
```

becomes

```
D:\MMOS2\INSTALL\MINSTALL.LOG
```

$(BOOT)

Replaces the startup (boot) drive letter of the operating system. (Do not append a colon.) For example:

```
$(BOOT)\OS2\NEW.SYS
```

becomes

```
C:\OS2\NEW.SYS
```

where *C:* is the drive on which OS/2 is installed.

Note: Using multiple macros is supported.

INI Change Control Files

You can use an INI change control file to do the following:

- Define folders and add them to the system
- Define programs and add them to a folder
- Install subsystems such as media control drivers, stream handlers, and I/O procedures. (See the *OS/2 Multimedia Subsystem Programming Guide* for further details.)

The following WPObject structures call the OS/2 WinCreateObject function, which adds an icon and title to the desktop. This structure indirectly changes the OS2.INI file. See the *OS/2 Presentation Manager Programming Reference* for object class definitions and supported keywords for the object class you are creating.

Defining Folders

Use the following structure to define a folder. Refer to the TLKBASE.SCR file located in the \TOOLKIT\SAMPLES\MM\SHORTCF subdirectory for an example.

```
WPObject =
 (
 WPClassName = "WPFolder"
 WPTitle = "title"
 WPSetupString = "ICONFILE=$(DEST)icon;OBJECTID=<folderobjid>"
 WPLocation = "<parentfolderobjid>"
 WPFlags = 0
 )
```

title Specifies the folder title to be displayed below the object.

icon Specifies the name of the icon to displayed on the desktop.

folderobjid Specifies the OS/2 unique object ID that is used to find this folder. This is used by the installation program to determine if this folder exists or not. It is also used in the *WPLocation* field of other *WPObject* definitions.

parentfolderobjid Specifies the folder object ID of the folder in which this folder is to be placed. For example, "<WP_DESKTOP>" is the object ID for the Workplace Shell desktop folder.

WPFlags = 0 Specifies creation control. These flags are also documented under WinCreateObject in the *OS/2 Presentation Manager Programming Reference* . already exist. Possible values include:

- 0 (CO_FAILIFEXISTS) - The folder is to be replaced if it exists, or added if it does not already exist.
- 1 (CO_REPLACEIFEXISTS) - The folder is to be replaced if it exists, or added if it does not already exist.
- 2 (CO_UPDATEIFEXISTS) - The folder is to be updated with the values supplied if it exists, or added using values supplied and defaults if it does not already exist.

In the following example, a folder called Multimedia Presentation Manager Toolkit/2 will be added to the desktop.

```
WPObject =
    (
    WPClassName   = "WPFolder"
    WPTitle       = "Multimedia Presentation\nManager Toolkit/2"
    WPSetupString = "ICONFILE=$(DEST)MMTOOLKT.ICO;OBJECTID=<MMPMTLK>"
    WPLocation    = "<WP_DESKTOP>"
    WPFlags = 2L
    )
```

Defining Programs

Use the following structure to define a program that is to be added to a folder. Refer to the TLKSTRN.SCR file located in the \TOOLKIT\SAMPLES\MM\SHORTCF subdirectory for an example.

```
WPObject =
(
WPClassName   = "WPProgram"
WPTitle       = "title"
WPSetupString = "EXENAME=path file;STARTUPDIR=dir;PROGTYPE=PM;
     ICONFILE=$(DEST)icon; [ASSOCTYPE=type;]
     [ASSOCFILTER=filter;] OBJECTID=<pgmobjid>"
WPLocation    = "<parentfolderobjid>"
WPFlags = 0
)
```

Note: The program setup information (`WPSetupString`) must be on one line, not two as shown in the previous figure.

title Specifies the title to be displayed below the object in the parent folder.

path Specifies a supported macro or the full path for the EXE file.

icon Specifies the name of the icon to be displayed on the desktop.

file Specifies the EXE file name.

dir Specifies the full path of the startup directory or the macro $(DIR)*#* (where *#* is a defined destination directory in CONTROL.SCR).

type Specifies one or more association types such as "Waveform." Multiple values are separated by commas.

filter Specifies one or more association filter types such as "*.WAV." Multiple values are separated by commas.

pgmobjid	Specifies the OS/2 unique object ID that is used to find this program. This is used by the installation program to determine if this program exists in the parent folder. It is not used in the *WPLocation* field of any *WPObject* definition.
parentfolderobjid	Specifies the folder object ID of the folder in which this program is to be placed.
WPFlags = 0	Specifies that the program is to be added to the folder only if it does not already exist.

Caption DLL

This section describes captioning functions and data structures provided in the Caption DLL found in the \TOOLKIT\SAMPLES\MM\CAPDLL subdirectory.

----

ccInitialize

----

ccInitialize - Syntax

This function creates a captioning window and returns the handle of that window to the application. This caption window is a child of *hwndParent*, and does not become visible until the Caption DLL is sent a CC_START command.

An application calls this function once, during its initialization (or before it wants to use the caption services).

```
#include <os2.h>

HWND     hwndParent;
ULONG    rc;

rc = ccInitialize(hwndParent);
```

----

ccInitialize Parameter - hwndParent

hwndParent (HWND) - input
    Parent window handle.

----

ccInitialize Return Value - rc

rc (ULONG) - returns
    If the function is successful, the caption window handle is returned. Possible return code inidicating type of failure:

CCERR_INVALID_WINDOW_HANDLE
The window handle passed was not valid.

----

ccInitialize - Parameters

hwndParent (HWND) - input
Parent window handle.

rc (ULONG) - returns
If the function is successful, the caption window handle is returned. Possible return code inidicating type of failure:

CCERR_INVALID_WINDOW_HANDLE
The window handle passed was not valid.

----

ccInitialize - Related Functions

- ccSendCommand
- ccTerminate

----

ccInitialize - Example Code

```
/*
 * Create the caption window and save the handle for further use.
 */
hwndCaption = ccInitialize ( (HWND) hwndMainDialogBox );
```

----

ccInitialize - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

----

ccSendCommand

----

ccSendCommand - Syntax

This function is used to control the captioning window once it has been created.

```
#include <os2.h>

USHORT    usMsg;
MPARAM    mp1;
MPARAM    mp2;
ULONG     rc;

rc = ccSendCommand(usMsg, mp1, mp2);
```

----

ccSendCommand Parameter - usMsg

usMsg (USHORT) - input

This is one of the following commands you want to send to the captioning window.

CC_START (1)

Use this command to start captioning. "Start captioning" means to display the window that was created in the ccInitialize command, and to begin scrolling the appropriate text file in synch with an audio file. Set *mp2* to point to a CC_START_PARMS structure.

CC_STOP (2)

Use this command to stop captioning. This command hides the caption window. Set *mp1* and *mp2* to zero.

CC_SET (3)

Use this command to set the status of the captioning system. The same items that can be queried can also be set. *mp2* should point to a CC_SET_PARMS structure. Note that these items can be queried and set even while captioning is going on.

CC_STATUS (4)

Use this command to query the status of the captioning system. Status items you can query and set include:

- Number of text columns
- Number of text rows
- Text color
- Background color
- Caption window X and Y coordinates

mp2 should point to a CC_STATUS_PARMS structure.

----

ccSendCommand Parameter - mp1

mp1 (MPARAM) - input

Caption window handle (returned from ccInitialize).

----

ccSendCommand Parameter - mp2

mp2 (MPARAM) - input
Data structure corresponding to the command message in *usMsg* .

----

ccSendCommand Return Value - rc

rc (ULONG) - returns
The CC_STOP return value is 0; the other commands have the following return codes indicating success or failure:

0
Returns if the function is successful.

CCERR_CANNOT_CREATE_BITMAP
Returned to CC_START or CC_SET; the bitmap cannot be created.

CCERR_FILE_FORMAT
Returned to CC_START; invalid file format was used.

CCERR_NO_DEVICE_NAME
Returned to CC_START; no device name was given.

CCERR_TOO_MANY_LINES
Returned to CC_START; caption file has more than 500 lines.

DOS error codes
Returned to CC_START.

OS/2 multimedia error codes
Returned to CC_START.

----

ccSendCommand - Parameters

usMsg (USHORT) - input
This is one of the following commands you want to send to the captioning window.

CC_START (1)
Use this command to start captioning. "Start captioning" means to display the window that was created in the ccInitialize command, and to begin scrolling the appropriate text file in synch with an audio file. Set *mp2* to point to a CC_START_PARMS structure.

CC_STOP (2)
Use this command to stop captioning. This command hides the caption window. Set *mp1* and *mp2* to zero.

CC_SET (3)
Use this command to set the status of the captioning system. The same items that can be queried can also be set. *mp2* should point to a CC_SET_PARMS structure. Note that these items can be queried and set even while

captioning is going on.

CC_STATUS (4)
Use this command to query the status of the captioning system. Status items you can query and set include:

- Number of text columns
- Number of text rows
- Text color
- Background color
- Caption window X and Y coordinates

mp2 should point to a CC_STATUS_PARMS structure.

mp1 (MPARAM) - input
Caption window handle (returned from ccInitialize).

mp2 (MPARAM) - input
Data structure corresponding to the command message in *usMsg* .

rc (ULONG) - returns
The CC_STOP return value is 0; the other commands have the following return codes indicating success or failure:

0
Returns if the function is successful.

CCERR_CANNOT_CREATE_BITMAP
Returned to CC_START or CC_SET; the bitmap cannot be created.

CCERR_FILE_FORMAT
Returned to CC_START; invalid file format was used.

CCERR_NO_DEVICE_NAME
Returned to CC_START; no device name was given.

CCERR_TOO_MANY_LINES
Returned to CC_START; caption file has more than 500 lines.

DOS error codes
Returned to CC_START.

OS/2 multimedia error codes
Returned to CC_START.

----

ccSendCommand - Related Functions

- ccInitialize
- ccTerminate

----

ccSendCommand - Example Code

```
/*
 * Close caption flag is ON.
 * Fill in the CC_START_PARMS structure and the call ccSendCommand
 * to make the captioning window visible.  The hwndOwner field holds
 * the window handle that we want the Caption DLL to send the position
 * change messages to, when it is done processing them.
 */
csp.pszDeviceName    = (PSZ)   "capsamp";     /* Alias name          */
csp.pszCaptionFile   = (PSZ)   "CAPSAMP._CC"; /* File name to use    */
csp.hwndOwner        =  hwnd;                 /* for position change */

ulReturn = ccSendCommand ( CC_START, MPFROMHWND(hwndCaption), &csp );
                                              /* Start captioning    */
```

----

ccSendCommand - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

----

ccTerminate

----

ccTerminate - Syntax

Call this function once at the end of the application (or when the application wishes to cease captioning). It closes the captioning system and releases any resources that were allocated on behalf of captioning.

```
#include <os2.h>

HWND    hwndCaption;

ccTerminate(hwndCaption);
```

----

ccTerminate Parameter - hwndCaption

hwndCaption (HWND)
Caption window handle.

----

ccTerminate - Return Value

This function does not return a value.

----

ccTerminate - Parameters

hwndCaption (HWND)
Caption window handle.

This function does not return a value.

----

ccTerminate - Related Functions

- ccInitialize
- ccSendCommand

----

ccTerminate - Example Code

```
/*
 * Close the captioning system. This will release all the
 * resources that were allocated.
 */
 ccTerminate(hwndCaption);
```

----

ccTerminate - Topics

Select an item:
Syntax
Parameters
Returns
Example Code
Related Functions
Glossary

CC_SET_PARMS

This data structure contains information for the ccSendCommand function when using the CC_SET message.

```
typedef struct _CC_SET_PARMS {
  ULONG     ulRows;
  ULONG     ulBackgroundColor;
  ULONG     ulTextColor;
  ULONG     ulXposition;
  ULONG     ulYposition;
  ULONG     ulColumns;
} CC_SET_PARMS;

typedef CC_SET_PARMS FAR *PCC_SET_PARMS;
```

CC_SET_PARMS Field - ulRows

ulRows (ULONG)
This is item is set to the desired number of text rows to display in the caption window.

CC_SET_PARMS Field - ulBackgroundColor

ulBackgroundColor (ULONG)
This item is set to the desired color index of the background color in the caption window.

CC_SET_PARMS Field - ulTextColor

ulTextColor (ULONG)
This item is set to the desired color index of the text color in the caption window.

CC_SET_PARMS Field - ulXposition

ulXposition (ULONG)
This item is set to the desired X-coordinate position of the caption window.

CC_SET_PARMS Field - ulYposition

ulYposition (ULONG)
This item is set to the desired Y-coordinate position of the caption window.

CC_SET_PARMS Field - ulColumns

ulColumns (ULONG)
This item is set to the desired number of columns to display in the caption window.

CC_START_PARMS

This data structure contains information for the ccSendCommand function when using the CC_START message.

```
typedef struct _CC_START_PARMS {
  HWND     hwndOwner;
  PSZ      pszDeviceID;
  PSZ      CaptionFile;
} CC_START_PARMS;

typedef CC_START_PARMS FAR *PCC_START_PARMS;
```

CC_START_PARMS Field - hwndOwner

hwndOwner (HWND)
This specifies the handle of the user window. If this parameter contains a window handle, then the captioning system will send MM_MCIPOSITIONCHANGE messages to the window when it is through with them. If this handle is NULL, then the application will not receive any MM_MCIPOSITIONCHANGE messages.

CC_START_PARMS Field - pszDeviceID

pszDeviceID (PSZ)
This is the ID of the device playing the audio which is to be captioned. If this parameter is not NULL, then the Caption DLL will request MM_MCIPOSITIONCHANGE messages from this device. If this parameter is NULL, then an error will be returned to the application.

CC_START_PARMS Field - CaptionFile

CaptionFile (PSZ)
This is a pointer to a NULL terminated string which contains the name of the caption file. If this parameter is NULL, or if the file cannot be opened, then an error will be returned to the application.

CC_STATUS_PARMS

This structure contains information for the ccSendCommand function when using the CC_STATUS message.

```
typedef struct _CC_STATUS_PARMS {
  ULONG     ulItem;
  ULONG     ulReturn;
} CC_STATUS_PARMS;

typedef CC_STATUS_PARMS FAR *PCC_STATUS_PARMS;
```

CC_STATUS_PARMS Field - ulItem

ulItem (ULONG)
This is the item whose status is being queried. It must be set to one of the following values:

CC_STATUS_TEXT_COLUMNS (1)
Columns of text in the caption window.

CC_STATUS_TEXT_ROWS (2)
Rows of text in the caption window.

CC_STATUS_TEXT_COLOR (3)
Color index of the text color.

CC_STATUS_BACKGROUND_COLOR (4)
Color index of the background color of the text window.

CC_STATUS_X_POSITION (5)
X-coordinate of the captioning window relative to the parent window.

CC_STATUS_Y_POSITION (6)
Y-coordinate of the captioning window relative to the parent window.

CC_STATUS_PARMS Field - ulReturn

ulReturn (ULONG)
This field contains the return value of the queried item.

BOOL

Boolean.

Valid values are:

- FALSE, which is 0
- TRUE, which is 1

```
typedef unsigned long BOOL;
```

HWND

Window handle.

```
typedef LHANDLE HWND;
```

MPARAM

A 4-byte message-dependent parameter structure.

```
typedef VOID *MPARAM;
```

Certain elements of information, placed into the parameters of a message, have data types that do not use all four bytes of this data type. The rules governing these cases are:

BOOL	The value is contained in the low word and the high word is 0.
SHORT	The value is contained in the low word and its sign is extended into the high word.
USHORT	The value is contained in the low word and the high word is 0.
NULL	The entire four bytes are 0.

The structure of this data type depends on the message. For details, see the description of the particular message.

PSZ

Pointer to a null-terminated string.

If you are using C++ **, you may need to use PCSZ.

```
typedef unsigned char *PSZ;
```

SHORT

Signed integer in the range -32 768 through 32 767.

```
#define SHORT short
```

ULONG

32-bit unsigned integer in the range 0 through 4 294 967 295.

```
typedef unsigned long ULONG;
```

USHORT

Unsigned integer in the range 0 through 65 535.

```
typedef unsigned short USHORT;
```

VOID

A data area of undefined format.

```
#define VOID void
```

High-Level Service API

This section describes audio-enabling macros and the high-level service routines designed to simplify writing macros. The service routines are valuable for many applications and are needed for information presentation facility (IPF) support.

The five functions (high-level service API) are: mciPlayFile, mciRecordAudioFile, mciPlayResource, mmioFindElement, and mmioRemoveElement. These functions are located in MCIAPI.DLL. Define INCL_MACHDR and include OS2ME.H in any source that needs access to the high-level service API. The *OS/2 Multimedia Programming Reference* provides a detailed description of these functions.

Note: These functions are also provided as 16-bit functions. The 16-bit functions include "16" in the function name as follows:

- mci16PlayFile
- mci16PlayResource
- mci16RecordAudioFile
- mmio16FindElement
- mmio16RemoveElement

The mciPlayFile function plays a multimedia file or waveaudio bundle element without displaying a device control panel. The mciPlayResource function operates much the same as mciPlayFile, except that it plays the multimedia object that is stored in a program resource and is intended for IPF support, not audio macros. The function loads the resource using DosGetResource, creates a MMIO memory file, and opens the default device to play the memory file.

The mciRecordAudioFile function provides a very simple recorder window allowing the recording of audio annotations. The mciRecordAudioFile function does not return until the recording is complete and the window is closed. This function creates the file if it does not exist. The mciRecordAudioFile records digital audio data from the microphone input of the default waveaudio device. When the file is initially created the settings are PCM, 11 kHz, mono, and the default sample size of the audio adapter in use. All play and record operations are performed from beginning to end.

The mmioRemoveElement function removes an element from a compound file and the mmioFindElement function enumerates elements in a compound file. For more information on the compound file structure, see RIFF Compound File Overview.

Creating an Audio-Enabled Macro for an Application

Many applications provide macro languages for extending the application's functionality. This is a possible way to integrate multimedia (usually audio) into existing applications. The basic function of an audio macro would be to record audio clips (annotations) and play them. For example, an audio clip can be associated with a location or object in a document such as a spreadsheet or worksheet.

Macros can use compound files to reduce the number of files in the system. Compound files provide a convenient way of grouping multimedia objects that are used together. The compound file structure provides for direct access to multimedia data elements.

If you were to annotate the cells of a spreadsheet and store the audio clips in a single compound file, the cell labels can be used as element names for the elements in the compound file. Then the macro can call mciRecordAudioFile and mciPlayFile to record and play back the elements of the compound file. The compound file can have the same base name as the worksheet and a file extension of .BND. This naming convention identifies which worksheet the audio files belong to. The macro does not need to keep track of cells that are moved on a worksheet, only the cell name is necessary to access the corresponding audio element of a compound file.

The audio file is created when you record or save the audio file. If you exit from a worksheet and choose not to save changes, the audio file will still exist. There is no way to recover an audio file after it is deleted (using mmioRemoveElement).

If a macro uses compound files and elements are deleted with mmioRemoveElement, the size of the compound file does not change. By specifying the MMIO_RE_COMPACT flag, a macro requests that the compound file be compacted after the element is removed. This allows audio recording into an existing wave element that can exceed the current size of the element. The resulting compound file can be compacted to save disk space.

Creating a REXX Command File Using MCI String Commands

Audio functions can be called with REXX command files. The audio functions available are mciRxSendString(), mciRxInit(), mciRxExit(), mciRxGetError(), and mciRxGetDeviceID(). OS/2 multimedia provides examples of REXX command files; PLAY.CMD and RECORD.CMD demonstrate how to play and record audio files. *Multimedia with REXX*, an online reference provided with OS/2 multimedia, contains more detailed information on the REXX string interface.

Duet Player IPF Sample

The Streaming Device Duet Sample (DUET1) uses mciPlayFile to play audio help. When the application receives an HM_INFORM message, mciPlayFile is issued as shown in the following code fragment.

```
   case HM_INFORM:
/*
 * The user has selected the "Play Audio Help" selection in the
 * IPF help panel.
 *
 * To initiate the playing of audio help, we need to issue
 * mciPlayFile.
 *
 * Note that we assume the HM_INFORM message came from the "Play
 * Audio Help" selection since it is the only :link. with an inform
 * reftype in the .ipf file.  If there were more, we would have to
 * check the resource identifier to determine for which selection
 * this message was generated.
 */
```

```
/*
 * Load the name of the audio help file from the resource.
 */
WinLoadString( hab,
               (HMODULE) NULL,
               IDS_HELP_WAVEFILE,
               (SHORT) sizeof( achAudioHelpFile),
               achAudioHelpFile);

ulError = mciPlayFile( (HWND)NULL,            /* Ignore owner
                                                 window            */
                       (PSZ)achAudioHelpFile, /* Audio file
                                                 to play           */
                       MCI_ASYNC,             /* Command processed
                                                 asynchronously    */
                       (PSZ)NULL,             /* Ignore title      */
                       (HWND)NULL );          /* Ignore viewport
                                                 window            */
if (ulError)
{
   ShowMCIErrorMessage( ulError);
}
return( (MRESULT) 0);
```

Notices

August 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

This publication was produced in the United States of America. IBM may not offer the products, services, or features discussed in this document in other countries, and the information is subject to change without notice. Consult your local IBM representative for information on the products, services, and features available in your area.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "(C) (your company name) (year). All rights reserved."

Note to U.S. Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Audio Visual Connection
Common User Access
C Set ++
CUA
IBM
Multimedia Presentation Manager/2
OS/2
Personal System/2
Presentation Manager
PS/2
Ultimotion
Workplace Shell
VisualAge

The following terms are trademarks of other companies:

DVI	Intel Corporation
Indeo	Intel Corporation
Motorola	Motorola, Inc.
NeXT	NeXT Computer, Inc.
Philips	Philips Electronics N.V.
Pioneer	Pioneer Electronic Corporation
Pro AudioSpectrum 16	Media Vision, Inc.
RealMagic	Sigma Designs, Inc.
Sony	Sony Corporation
Sound Blaster	Creative Technology Ltd.
Sun	Sun Microsystems, Inc.
WinTV	Hauppauge Computer Works, Inc.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

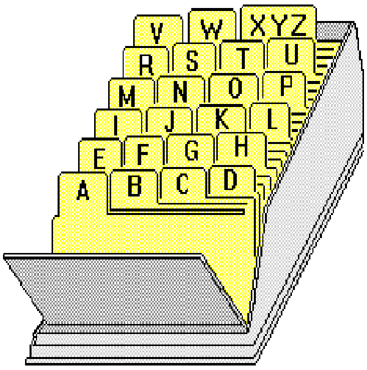
UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

Glossary

Select the starting letter of the glossary term you want to locate.



A

AB roll - Synchronized playback of two recorded video images so that one can perform effects, such as dissolves, wipes, or inserts, using both images simultaneously.

ACPA - Audio capture and playback adapter.

active matrix - A technology that gives every pel (dot) on the screen its own transistor to control it more accurately. (This allows for better contrast and less motion smearing.)

adaptive differential pulse code modulation - A bit-rate reduction technique where the difference in pulse code modulation samples are not compressed before being stored.

ADC - Analog-to-digital converter.

ADPCM - Adaptive differential pulse code modulation.

aliasing - The phenomenon of generating a false (alias) frequency, along with the correct one, as an artifact of sampling a signal at discrete points. In audio, this produces a "buzz." In imagery, this produces a jagged edge, or stair-step effect. See also *moire* .

all points addressable (APA) - In computer graphics, pertaining to the ability to address and display or not display each picture element on a display surface.

AM - Animation metafile.

ambience - In audio, the reverberation pattern of a particular concert hall, or listening space.

ambient noise - In acoustics, the noise associated with a particular environment, usually a composite of sounds from many distant or nearby sources.

American National Standards Institute (ANSI) - An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

AMF - Animation metafile format.

amp - See *amplifier* .

amplifier - (1) A device that increases the strength of input signals (either voltage or current). (2) Also referred to as an *amp* .

amp-mixer - (1) A combination amplifier and mixer that is used to control the characteristics of an audio signal from one or more audio sources. (2) Also referred to as an amplifier-mixer.

analog - Pertaining to data consisting of continuously variable physical quantities. Contrast with *digital* .

analog audio - Audio in which all information representing sounds is stored or transmitted in a continuous-scale electrical signal, such as line level audio in stereo components. See also *digital audio.*

analog-to-digital converter (ADC) - A functional unit that converts data from an analog representation to a digital representation. (I) (A)

analog video - Video in which all the information representing images is in a continuous-scale electrical signal for both amplitude and time. See also *digital video.*

analog video overlay - See *overlay* .

anchor - An area of a display screen that is activated to accept user input. Synonymous with *hot spot, touch area,* and *trigger* .

animate - Make or design in such a way as to create apparently spontaneous, lifelike movement.

animated - Having the appearance of something alive.

animated screen capture - Recording a computing session for replay on a similar computer with voice annotation. (An example is sending a spreadsheet with an accompanying screen recording as an explanation and overview.)

animatic - A limited animation consisting of artwork shot on film or videotape and edited to serve as an on-screen storyboard.

animation metafile - A compound file format, the elements of which are the frames of animation themselves. These frames are stored sequentially so that they can be played back in time by streaming to the video agent.

animation metafile format (AMF) - The file format used to store animated frame sequences.

annotation - The linking of an object with another, where the second contains some information related to the first. For example, an audio annotation of a spreadsheet cell might contain verbal explanation about the contents of the cell.

ANSI - American National Standards Institute.

anthropomorphic software agent - The concept of a simulated agent, seemingly living inside the computer, that talks to and listens to the user, and then acts for the user on command.

antialiasing - (1) In imagery, using several intensities of colors (a ramp) between the color of the line and the background color to create the effect of smoother curves and fewer jagged edges on curves and diagonals. (2) In imagery or audio, removing aliases by eliminating frequencies above half the sample frequencies.

AOCA - Audio Object Content Architecture.

APA - All points addressable.

APA graphics - All Points Addressable graphics. See *bitmap graphics* .

API - Application programming interface.

application programming interface (API) - A functional interface supplied by the operating system or an IBM separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application-supplied video window - (1) An application can specify to MMPM/2 that it wants video played in a specific window (controlled by the application) instead of the default window (controlled by MMPM/2). The application supplied video window can be used to implement advanced features not supported by the default video window. (2) Also referred to as an *alternate video window* .

artifact - A product resulting from human activity; in computer activity, a (usually unwanted) by-product of a process.

aspect ratio - (1) On a display screen, the ratio of the maximum length of a display line to the maximum length of a display column. (2) The ratio of height to width. (This term applies to areas or individual pels.) Refer to *enhanced graphics adapter* and *video graphics adapter* .

asymmetric video compression - In multimedia applications, the use of a powerful computer to compress a video for mastering so that a less powerful (less expensive) system is needed to decompress it. Contrast with *symmetric video compression* .

audible cue - A sound generated by the computer to draw a user's attention to, or provide feedback about, an event or state of the computer. Audible cues enhance and reinforce visible cues.

audio - Pertaining to the portion of recorded information that can be heard.

audio attribute control - Provides access to and operation of the standard audio attributes: mute, volume, balance, treble, and bass. All device communication and user interface support is handled by the control.

audio attributes - Refers to the standard audio attributes: mute, volume, balance, treble and bass.

audio clip - A section of recorded audio material.

Audio Object Content Architecture - A data format for multimedia products.

audio processing - Manipulating digital audio to, for example, edit or create special effects.

audio segment - A contiguous set of recorded data from an audio track. An audio segment might or might not be associated with a video segment.

audio track - (1) The audio (sound) portion of the program. (2) The physical location where the audio is placed beside the image. (A system with two audio tracks can have either stereo sound or two independent audio tracks.) (3) Synonym for *sound track* .

audiovisual - A generic term referring to experiences, equipment, and materials used for communication that make use of both hearing and sight.

Audio Visual Connection* (AVC) - An authoring system used on an IBM PS/2* to develop and display audiovisual productions.

audiovisual computer program - A computer program that makes use of both hearing and sight.

authoring - A structured approach to combining all media elements within an interactive production, assisted by computer software designed for this purpose.

authoring system - A set of tools used to create an interactive multimedia application without implementing formal programming.

AVI file format - The Audio/Video Interleaved (AVI) file format is the standard file format used to support software motion video. AVI files can contain multiple streams (tracks) of data (for example, a video and an audio stream). The streams are interleaved to improve access times during playback. The present implementation is limited to a single video stream and a single, optional, audio stream.

B

background image - The part of a display image, such as a form overlay, that is not changed during a particular sequence of transactions. Contrast with *foreground image* .

background music - In videotaping, music that accompanies dialog or action.

balance - For audio, refers to the relative strength of the left and right channels. A balance level of 0 is left channel only. A balance level of 100 is right channel only.

basic input/output system (BIOS) - In an IBM personal computer, microcode that controls basic hardware operations, such as interactions with diskette drives, hard disk drives, and the keyboard. (For example, the IBM Enhanced Graphics Adapter has an addressable BIOS, located on the adapter itself, that is used to control the IBM InfoWindow* graphics functions.)

beta format - A consumer and industrial 0.5-inch tape format.

BG - Script abbreviation for *background* .

BIOS - Basic input/output system.

bitmap - A coded representation in which each bit, or group of bits, represents or corresponds to an item, for example, a configuration of bits in main storage in which each bit indicates whether a peripheral device or a storage block is available or in which each group of bits corresponds to one pel of a display image.

bit-block transfer - Transfer of a rectangular array of bitmap data.

bitblt - Bit-block transfer. Synonym for *blit* .

bitmap graphics - A form of graphics whereby all points on the display are directly addressable. See also *all points addressable* .

blit - Synonym for *bitblt* .

blitter - Hardware that performs bit-block transfer operations.

BND - (1) An internal I/O procedure, provided by the MMPM/2 system, that supports RIFF compound files (commonly called *bundle files*).

(2) The four-character code (*FOURCC*) of a bundle file. (3) See also *RIFF compound file*.

BND file - A RIFF compound file.

BND IOProc - An internal I/O procedure, provided by the MMPM/2 system, that supports the elements in a RIFF compound file. See also *CF IOProc*.

boom - A long, relatively horizontal supporting brace used for holding a microphone or camera; sometimes used to refer to the machinery that supports the camera and allows it to move while shooting.

brightness - Refers to the level of luminosity of the video signal. A brightness level of 0 produces a maximally white signal. A brightness level of 100 produces a maximally black signal.

buffer - A portion of storage used to hold input or output data temporarily.

bundle file (BND) - (1) A file that contains many individual files, called *file elements*, bound together. The MMIO file manager provides services to locate, query, and access file elements in a bundle file. (2) A RIFF compound file.

bus - A facility for transferring data between several devices located between two end points, only one device being able to transmit at a given moment.

buy - (1) In videotaping, footage that is judged acceptable for use in the final video. (2) Synonym for *keeper*.

C

CAI - Computer-assisted instruction. Synonym for *CBT*.

calibration - The adjustment of a piece of equipment so that it meets normal operational standards. (For example, for the IBM InfoWindow system, calibration refers to touching a series of points on the screen so that the system can accurately determine their location for further reference.)

camcorder - A compact, hand-held video camera with integrated videotape recorder.

capture - To take a snapshot of motion video and retain it in memory. The video image may then be saved to a file or restored to the display.

cast animation - (1) A sequence of frames consisting of manipulations of graphical objects. (2) The action of bringing a computer program, routine, or subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (3) See also *frame animation*.

CAV - Constant angular velocity.

CBT - Computer-based training. Synonym for *CAI*.

CCITT - Comite Consultatif International Telegraphique et Telephonique. The International Telegraph and Telephone Consultative Committee.

CD - Compact disc.

CD-DA - Compact disc, digital audio.

CD-I - Compact Disc-Interactive.

CD-ROM - Compact disc, read-only memory.

CD-ROM XA - Compact disc, read-only memory extended architecture.

cel - A single frame (display screen) of an animation. (The term originated in cartooning days when the artist drew each image on a sheet of celluloid film.)

CF IOProc - An internal I/O procedure, provided by the MMPM/2 system, that supports RIFF compound files. The CF IOProc operates on the entire compound file (rather than on the elements in a RIFF compound file, as with the BND IOProc). The CF IOProc is limited to operations required by the system to ensure storage system transparency at the application level. See also *BND IOProc*.

CGA - Color graphics adapter.

CGRP - Compound file resource group.

channel mapping - The translation of a MIDI channel number for a sending device to an appropriate channel for a receiving device.

channel message - A type of non-SysEx MIDI message that has a channel identifier in it, implying that these messages are specific to one channel.

check disc - A videodisc produced from the glass master that is used to check the quality of the finished *interactive program* .

chord - To press more than one button at a time on a pointing device.

chroma-key color - The specified first color in a combined signal. See also *chroma-keying* .

chroma-keying - Combining two video signals that are in sync. The combined signal is the second signal whenever the first is of some specified color, called the chroma-key color, and is the first signal otherwise. (For example, the weatherman stands in front of a blue background-blue is the chroma-key color.) At home, the TV viewer sees the weather map in place of the chroma-key color, with the weatherman suspended in front.

chroma signal - The portion of image information that provides the color (hue and saturation).

chrominance - The difference between a color and a reference white of the same luminous intensity.

chunk - (1) The basic building block of a RIFF file. (2) A RIFF term for a formalized data area. There are different types of chunks, depending on the chunk ID. (3) See *LIST chunk* and *RIFF chunk* .

chunk ID - A four-character code (FOURCC) that identifies the representation of the chunk data.

circular slider control - A knob-like control that performs like a control on a TV or stereo.

circular slider knob - A knob-like dial that operates like a control on a television or stereo.

class - (1) A categorization or grouping of objects that share similar behaviors and characteristics. Synonym for *object class.* (2) See *node class* . (RTMIDI-specific term)

click - To press and release a button on a pointing device without moving the pointer off the choice. See *double-click* . See also *drag select* .

clip - A section of recorded, filmed, or videotaped material. See also *audio clip* and *video clip* .

closed circuit - A system of transmitting television signals from a point of origin to one or many restricted destination points specially equipped to receive the signals.

close-up - In videotaping, the picture obtained when the camera is positioned to show only the head and shoulders of a subject; in the case of an object, the camera is close enough to show details clearly. See also *extreme close-up* .

CLP - Common loader primitive.

CLUT - Color look-up table. Synonym for *color palette* .

CLV - Constant linear velocity.

CODEC - compressor/decompressor (CODEC) - An algorithm implemented either in hardware or software that can either compress or decompress a data object. For example, a CODEC can compress raw digital images into a smaller form so that they use less storage space. When used in the context of playing motion video, decompressors reconstruct the original image from the compressed data. This is done at a high rate of speed to simulate motion.

collaborative document production - A system feature that provides the ability for a group of people to manage document production.

collision - An unwanted condition that results from concurrent transmissions on a channel. (T) (For example, an overlapping condition that occurs when one sprite hides another as it passes over it.)

color cycling - An animation effect in which colors in a series are displayed in rapid succession.

color graphics adapter (CGA) - An adapter that simultaneously provides four colors and is supported on all IBM Personal Computer and Personal System/2* models.

colorization - The color tinting of a monochrome original.

color palette - (1) A set of colors that can be displayed on the screen at one time. This can be a standard set used for all images or a set that can be customized for each image. (2) Synonym for *CLUT* . (3) See also *standard palette* and *custom palette* .

common loader primitive - A system service that provides a high-level interface to hardware-specific loaders.

common user access (CUA) - (1) Guidelines for the dialog between a human and a workstation or terminal. (2) One of the three SAA architectural areas.

compact disc (CD) - A disc, usually 4.75 inches in diameter, from which data is read optically by means of a laser.

compact disc, digital audio (CD-DA) - The specification for audio compact discs. See also *Redbook audio* .

Compact Disc-Interactive (CD-I) - A low-cost computer, being developed by N.V. Phillips (The Netherlands) and Sony (Japan), that plugs into standard television sets to display text and video stored on compact discs.

compact disc, read-only memory (CD-ROM) - High-capacity, read-only memory in the form of an optically read compact disc.

compact disc, read-only memory extended architecture (CD-ROM XA) - An extension to CD-ROM supporting additional audio and video levels for compression and interlacing of audio, video, and digital data.

component video - A video signal using three signals, one of which is luminance, and the other two of which are the color vectors. See also *composite video* and *S-video* .

composed view - A view of an object in which relationships of the parts contribute to the overall meaning. Composed views are provided primarily for data objects.

composite - The combination of two or more film, video, or electronic images into a single frame or display. See also *composite video* .

composite monitor - A monitor that can decode a color image from a single signal, such as NTSC or PAL. Contrast with *RGB* .

composite object - An object that contains other objects. For example, a document object that contains not only text, but graphics, audio, image, and/or video objects, each of which can be manipulated separately as an individual object.

composite video - A single signal composed of chroma, luminance, and sync. NTSC is the composite video that is currently the U.S. standard for television. See also *component video* and *S-video* .

compound device - A multimedia device model for hardware that requires additional data objects, referred to as data elements, before multimedia operations can be performed.

compound file - A file that contains multiple file elements.

compound file resource group (CGRP) - A RIFF chunk that contains all the compound file elements, concatenated together.

compound file table of contents (CTOC) - A *RIFF chunk* that indexes the CGRP chunk, which contains the actual multimedia data elements. Each entry contains the name of, and other information about, the element, including the offset of the element within the CGRP chunk. All the *CTOC* entries of a table are of the same length and can be specified when the file is created.

compound message - A structure which combines a time stamp, a source instance identifier, a track number, and a MIDI message. Each of these fields is 32 bits, so the structure is 16 bytes in length. (RTMIDI-specific term)

computer-animated graphics - Graphics animated by using a computer, compared to using videotape or film.

computer-assisted instruction (CAI) - (1) A data processing application in which a computing system is used to assist in the instruction of students. The application usually involves a dialog between the student and a computer program. An example is the OS/2 tutorial. (2) Synonym for *computer-based training* .

computer-based training (CBT) - Synonym for *computer-assisted instruction* .

computer-controlled device - An external video source device with frame-stepping capability, usually a videodisc player, whose output can be controlled by the multimedia subsystem.

conforming - Performing final editing on film or video using an offline edited master as a guide.

connection - The establishment of the flow of information from a connector on one device to a compatible connector on another device. A connection can be made that is dependent on a physical connection, for example the attachment of a speaker to an audio adapter with a speaker wire. A connection can also be made that is completely internal to the PC, such as the connection between the waveaudio media device and the ampmix device. See also *connector* .

connector - A software representation of the physical way in which multimedia data moves from one device to another. A connector can have an external representation, such as a headphone jack on a CD-ROM player. A connector can also have an internal representation, such as the flow of digital information into an audio adapter. See also *connection* .

constant angular velocity (CAV) - Refers to both the format of data stored on a videodisc and the videodisc player rotational characteristics. CAV videodiscs contain 1 frame per track. This allows approximately 30 minutes of playing time per videodisc side. CAV videodisc players spin at a constant rotational speed (1800 rpm for NTSC or 1500 rpm for PAL) and play 1 frame per disc revolution. CAV players support *frame-accurate searches* . See also *constant linear velocity* .

constant linear velocity (CLV) - Refers to both the format of data stored on a videodisc and the videodisc player characteristics. CLV videodiscs contain 1 frame on the innermost track and 3 frames of data on the outermost track. This allows approximately 1 hour of playing time per videodisc side. CLV videodisc players vary the rotational speed from approximately 1800 rpm at the inner tracks to 600 rpm at the outer tracks (for NTSC).

Currently, few CLV players support *frame-accurate searches* They only support search or play to within one second (30 frames for NTSC or 25 frames for PAL). See also *constant angular velocity* .

container - An object whose specific purpose is to hold other objects. A folder is an example of a container object.

contents view - A view of an object that shows the contents of the object in list form. Contents views are provided for container objects and for any object that has container behavior, for example, a device object such as a printer.

continuity - In videotaping, consistency maintained from shot to shot and throughout the take. For example, a switch that is on in one shot should not be off in the next unless it was shown being turned off.

continuous media object - A data object that varies over time; a stream-oriented data object.   Examples include audio, animation, and video.

contrast - The difference in brightness or color between a display image and the area in which it is displayed. A contrast level of 0 is minimum difference. A contrast level of 100 is maximum difference.

control - A visual user interface component that allows a user to interact with data.

coordinate graphics - (1) Computer graphics in which display images are generated from display commands and coordinate data. (2) Contrast with *raster graphics* . (3) Synonym for *line graphics* .

crop - To cut off; to trim (for example, a tape).

crossfade - Synonym for *dissolve* .

cross-platform - Used to describe applications that are operable with more than one operating system.

cross-platform transmission - Electronic transmission of information (such as mail) between incompatible operating systems.

CTOC - Compound file table of contents.

CU - Script abbreviation for *close-up* .

CUA - Common User Access.

cue point - A point that the system recognizes as a signal that may be acted upon.

custom palette - (1) A set of colors that is unique to one image or one application. (2) See also *standard palette* and *color palette* .

cut - The procedure of instantly replacing a picture from one source with a picture from another. (This is the most common form of editing scene to scene.)

D

DAC - Digital-to-analog converter.

data object - In an application, an element of a data structure (such as a file, an array, or an operand) that is needed for program execution and that is named or otherwise specified by the allowable character set of the language in which the program is coded.

data stream - All data transmitted through a data channel.

data streaming - Real-time, continuous flowing of data.

DCP - See *device control panel* .

decode - (1) To convert data by reversing the effect of previous encoding. (2) To interpret a code. (3) To convert encoded text into plaintext by means of a code system. (4) Contrast with *encode* .

default video window - (1) Refers to where video is displayed when an application does not indicate an application-defined window with the MCI_WINDOW message. This is provided by and managed for the application by MMPM/2. (2) See also *application-defined window* .

default window - See *default video window* .

delta frame - Refers to one or more frames occurring between reference frames in the output stream. Unlike a reference frame, which stores a complete image, a delta frame stores only the changes in the image from one frame to the next. See *reference frame* .

destination rectangle - An abstract region which defines the size of an image to be created when recording images for software motion video playback. The ratio of this rectangle's size to that of the source rectangle determines the scaling factor to be applied to the video.

destination window - See *destination rectangle* .

device capabilities - The functionality of a device, including supported component functions.

device context - The device status and characteristics associated with an opened instance of an Media Control Interface device.

device control panel (DCP) - An integrated set of controls that is used to control a device or media object (such as by playing, rewinding, increasing volume, and so on).

device controls - See *standard multimedia device controls* .

device driver - (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, a videodisc player, or a CD drive.

device element - A data object, such as a file, utilized by a compound device.

device object - An object that provides a means for communication between a computer and the outside world. A printer is an example of a device object.

device sharing - (1) The ability to share a device among many different applications simultaneously. If a device is opened shareable, the device context will be saved by the operating system when going from one application to another application. (2) Allowing a device context to be switched between Media Control Interface devices.

device-specific format - The storage or transmission format used by a device, especially if it is different from an accepted standard.

dialog - In an interactive system, a series of related inquiries and responses similar to a conversation between two people.

digital - (1) Pertaining to data in the form of numeric characters. (2) Contrast with *analog* .

digital audio - (1) Material that can be heard that has been converted to digital form. (2) Synonym for *digitized audio* .

digital signal processor (DSP) - A high-speed coprocessor designed to do real-time manipulation of signals.

digital video - (1) Material that can be seen that has been converted to digital form. (2) Synonym for *digitized video* .

digital video effects (DVE) - An online editing technique that manipulates on-screen a full video image; activity for creating sophisticated transitions and special effects. Digital video effects (DVE) can involve moving, enlarging, or overlaying pictures.

Digital Video Interactive (DVI) - A system for bringing full-screen, full-motion television pictures and sound to a regular PC. DVI is a chip set and uses delta compression; that is, only the image-to-image changes in each frame are saved rather than the whole frame. Data (video footage) is compressed into a form that reduces memory requirements by factors of 100 or greater. This compressed data is stored on optical discs and can be retrieved at a rate of 30 frames per second. (The DVI technology was developed by RCA and then sold to Intel. IBM has chosen this technology for future use in the PS/2.)

digital-to-analog converter (DAC) - (1) A functional unit that converts data from a digital representation to an analog representation. (2) A device that converts a digital value to a proportional analog signal.

digitize - To convert an analog signal into digital format. (An analog signal during conversion must be *sampled* at discrete points and quantized to discrete numbers.)

digitized audio - Synonym for *digital audio* .

digitized video - Synonym for *digital video* .

digitizer - A device that converts to digital format any image captured by the camera.

direct manipulation - A set of techniques that allow a user to work with an object by dragging it with a pointing device or interacting with its pop-up menu.

direct memory access - The transfer of data between memory and input and output units without processor intervention.

direct-read-after-write (DRAW) disc - A videodisc produced directly from a videotape, one copy at a time. A DRAW disc usually is used to check program material and author applications before replicated discs are available.

disc - Alternate spelling for *disk* .

discard stop - In data streaming, requests that the data stream be stopped and the data remaining in the stream buffers be discarded.

disk - A round, flat, data medium that is rotated in order to read or write data.

display image - A collection of display elements or segments that are represented together at any one time on a display surface. See also *background image* and *foreground image* .

dissolve - To fade down one picture as the next fades up. Synonym for *crossfade* .

dithering - When different pixels in an image are prebiased with a varying threshold to produce a more continuous gray scale despite a limited palette. This technique is used to soften a color line or shape. This technique also is used for alternating pixel colors to create the illusion of a third color.

DLL - Dynamic-link library.

dolly - A wheeled platform for a camera; a camera movement where the tripod on which the camera is mounted physically moves toward or away from the subject.

DOS IOProc - An internal I/O procedure, provided by the MMPM/2 system, that supports DOS files.

double-click - In SAA Advanced Common User Access, to press and release a mouse button twice within a time frame defined by the user, without moving the pointer off the choice. See *click* . See also *drag select* .

drag select - In SAA Advanced Common User Access, to press a mouse button and hold it down while moving the pointer so that the pointer travels to a different location on the screen. Dragging ends when the mouse button is released. All items between the button-down and button-up points are selected. See also *click* , *double-click* .

DRAW disc - Direct-read-after-write disc.

drop-frame time code - A nonsequential time code used to keep tape time code matched to real time. Must not be used in tapes intended for videodisc mastering.

DSP - Digital signal processor.

DTMF - Dual-tone modulation frequency.

dual plane video system - Refers to when graphics from the graphics adapter are separate from the analog video. That is, there is a separate graphics plane and video plane. The analog video appears behind the graphics, showing through only in the areas that are transparent. Since graphics and video are separate, capturing the graphics screen will only obtain graphics, and capturing the video screen will only obtain video. This is also true for restoring images. See also *single plane video system* .

dual-state push button - A push button that has two states, in and out. It is used for setting and resetting complementary states, such as Mute and Unmute.

dual-tone modulation frequency (DTMF) - Pushbutton phone tones.

dub - To copy a tape; to add (sound effects or new dialog) to a film; to provide a new audio track of dialog in a different language. (Often used with "in" as "dub in".)

DVE - Digital video effects.

DVI - Digital Video Interface

dynamic icon - An icon that changes to convey some information about the object that it represents. For example, a folder icon can show a count, indicating the number of objects contained within the folder. Also, a tape player icon can show an animation of turning wheels to indicate that the machine playing.

dynamic linking - In the OS/2 operating system, the delayed connection of a program to a routine until load time or run time.

dynamic link library (DLL) - A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic resource allocation - An allocation technique in which the resources assigned for execution of computer programs are determined by criteria applied at the moment of need. (I) (A)

dynamic resource - A multimedia program unit of data that resides in system memory. Contrast with *static resource* .

E

earcon - An icon with an audio enhancement, such as a ringing telephone.

ECB - Event control block.

ECU - Script abbreviation for *extreme close-up* .

edit decision list (EDL) - Synonym for *edit list* .

edit list - A list of the specific video footage, with time-code numbers, that will be edited together to form the program. It is completed during the offline edit and used during the online edit. Synonym for *edit decision list (EDL).*

edit master - The final videotape from which all copies are made. See also *glass master* .

editing - Assembling various segments into the composite program.

EDL - Edit decision list.

EGA - Enhanced graphics adapter.

element - (1) A file or other stored data item. (2) An individual file that is part of a RIFF compound file. An element of a compound file also could be an entire RIFF file, a non-RIFF file, an arbitrary RIFF chunk, or arbitrary binary data. (3) The particular resource within a subarea that is identified by an element address.

emphasis - Highlighting, color change, or other visible indication of the condition of an object or choice and the effect of that condition on a user's ability to interact with that object or choice. Emphasis can also give a user additional information about the state of an object or choice.

encode - To convert data by the use of a code in such a manner that reconversion to the original form is possible. Contrast with *decode* .(T)

enhanced graphics adapter (EGA) - A graphics controller for color displays. The pel resolution of an enhanced graphics adapter is 3:4.

entry field - An area into which a user places text. Its boundaries are usually indicated.

erasable optical discs - Optical discs that can be erased and written to repeatedly.

establishing shot - In videotaping, a long shot used in the beginning of a program or segment to establish where the action is taking place and to give the sense of an environment.

EVCB - Event control block.

event - An occurrence of significance to a task; for example, the completion of an asynchronous operation, such as I/O.

event control block (ECB or EVCB) - A control block used to represent the status of an event.

event queue - In computer graphics, a queue that records changes in input devices such as buttons, valuators, and the keyboard. The event queue provides a time-ordered list of input events.

event semaphore - (1) Used when one or more threads must wait for a single event to occur. (2) A blocking flag used to signal when an event has occurred.

explicit event - An event supported by only some handlers, such as a custom event unique to a particular type of data.

EXT - Script abbreviation for *exterior* .

extended selection - A type of selection optimized for the selection of a single object. A user can extend selection to more than one object, if required. The two kinds of extended selection are contiguous extended selection and discontiguous extended selection.

extreme close-up - The shot obtained when the camera is positioned to show only the face or a single feature of the subject; in the case of an object, the camera is close enough to reveal an individual part of the object clearly.

F

facsimile machine - A functional unit that converts images to signals for transmission over a telephone system or that converts received signals back to images.

fade - To change the strength or loudness of a video or audio signal, as in "fade up" or "fade down."

fast threads - Threads created by an application that provide minimal process context, for example, just stack, register, and memory. With the reduced function, fast threads can be processed quickly.

FAX machine - Synonym for facsimile machine.

file cleanup - The removal of superfluous or obsolete data from a file.

file compaction - Any method of encoding data to reduce its required storage space.

file element - An individual file that is part of a RIFF compound file. An element of a compound file can also be an entire RIFF file, a non-RIFF file, an arbitrary RIFF chunk, or arbitrary binary data. See *media element* .

file format - A language construct that specifies the representation, in character form, of data objects in a file. For example, MIDI, M-Motion, or AVC.

file format handler - (1) I/O procedure. (2) Provides functions that operate on the media object of a particular data format. These functions include opening, reading, writing, seeking, and closing elements of a storage system.

file format IOProc - (1) An *installable I/O procedure* that is responsible for all technical knowledge of the format of a specific type of data, such as headers and data compression schemes. A file format IOProc manipulates multimedia data at the element level. A file format IOProc handles the element type it was written for and does not rely on any other file format IOProcs to do any processing. However, a file format IOProc might need to call a *storage system IOProc* to obtain data within a file containing multiple file elements. (2) See *IOProc* . (3) See also *storage system IOProc* .

filter - A certain type of node that modifies messages and forwards them. Filters are used to perform real-time processing of MIDI data. When a filter receives a message, it may perform some manipulation on it. It will then forward the message. (RTMIDI-specific term)

final script - The finished script that will be used as a basis for shooting the video. Synonym for *shooting script* .

first draft - A rough draft of the complete script.

first generation - In videotaping, the original or master tape; not a copy.

flashback - Interruption of chronological sequence by interjection of events occurring earlier.

flush stop - In data streaming, requests that the source stream handler be stopped but the target stream handler continue until the last buffer held at the time the stop was requested is consumed by the target stream handler.

flutter - A phenomenon that occurs in a videodisc freeze-frame when both video fields are not identically matched, thus creating two different pictures alternating every 1/60th of a second.

fly-by - Animation simulating a bird's-eye view of a three-dimensional environment.

fly-in - A DVE where one picture "flies" into another.

folder - A file used to store and organize documents or electronic mail.

footage - The total number of running feet of film used (as for a scene).

foreground image - The part of a display image that can be changed for every transaction. Contrast with *background image* .

form overlay - A pattern such as a report form, grid, or map used as background for a display image.

form type - A field in the first four bytes of the data field of a RIFF chunk. It is a four-character code identifying the format of the data stored in the file. A RIFF form is a chunk with a chunk ID of RIFF. For example, waveform audio files (WAVE files) have a form type of WAVE.

Format 0 MIDI file - All MIDI data is stored on a single track.

Format 1 MIDI file - All MIDI data is stored on multiple tracks.

forward - To re-transmit a message that was received. Each instance can have any number of links from it. When an instance receives a message, it may decide to send the same message along its links. This is known as forwarding. (RTMIDI-specific term)

four-character code (FOURCC) - A 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters (padded on the right with blank characters). Four-character codes are unique identifiers that represent the file format and I/O procedure.

FOURCC - Four-character code.

fps - Frames per second.

frame - In film, a complete television picture that is composed of two scanned fields, one of the even lines and one of the odd lines. In the NTSC system, a frame has 525 horizontal lines and is scanned in 1/30th of a second.

frame-step recording - Refers to the capturing of video and audio data frame by frame, from a computer-controlled, frame-steppable video source device, or a previously recorded AVI file.

frame-accurate searches - The ability of a videodisc player to play or search to specific frames on the videodisc via software or remote control. This capability is available on all CAV players, but currently only on a few CLV players. Most CLV players can only search or play to within one second (30 frames for NTSC or 25 frames for PAL).

frame animation - A process where still images are shown at a constant rate. See also *cast animation* .

frame grabber - A device that digitizes video images.

frame number - The number used to identify a frame. On videodisc, frames are numbered sequentially from 1 to 54,000 on each side and can be accessed individually; on videotape, the numbers are assigned by way of the SMPTE time code.

frame rate - The speed at which the frames are scanned-30 frames a second in NTSC video, 25 frames a second in PAL video, and 24 frames a second in *most* film.

A complete television picture frame is composed of two scanned fields, one of the even lines and one of the odd lines. In the NTSC system, a frame has 525 horizontal lines and is scanned in 1/30th of a second. In the PAL system, a frame has 625 horizontal lines and is scanned in 1/25th of a second.

freeze - Disables updates to all or part of the video buffer. The last video displayed remains visible. See also *unfreeze* .

freeze-frame - A frame of a motion-picture film that is repeated so as to give the illusion of a still picture.

full-frame time code - (1) A standardized method, set by the Society of Motion Picture and Television Engineers (SMPTE), of address coding a videotape. It gives an accurate frame count rather than an accurate clock time. (2) Synonym for *nondrop time code* .

full-motion video - Video playback at 30 frames per second for NTSC signals or 25 frames per second for PAL signals.

G

game port - On a personal computer, a port used to connect devices such as joysticks and paddles.

GDT - Global Descriptor Table.

general purpose interface bus - An adapter that controls the interface between the PC, Personal Computer XT*, or Personal Computer AT* and, for example, the InfoWindow display; also known as the IBM IEEE 488.

genlock - A device that comes on an adapter or plugs into a computer port and provides the technology to overlay computer-generated titles and graphics onto video images. It does this by phase-*lock* ing the sync *gen* eration of two video signals together so they can be merged. Genlock also converts a digital signal to NTSC or PAL format. (For example, flying logos and scrolling text on television shows are overlaid using a genlock.)

glass master - The final videodisc format from which copies are made.

Global Descriptor Table (GDT) - Defines code and data segments available to all tasks in an application.

GOCA - Graphic Object Content Architecture.

Graphic Object Content Architecture (GOCA) - (1) A data format for multimedia products. (2) A push button with graphic, two-state, and animation capabilities.

graphics overlay - The nature of dual plane video systems makes it possible to place graphics over video. Only the pels of the designated transparent color allows the video to show through. All other graphics pels appear on top of the video. Note that the video still exists in the video buffer under the non-transparent graphics pels.

graphics plane - In a dual plane video system, the graphics plane contains material drawn or generated by the graphics adapter. The graphics plane will be combined with the video plane to create an entire display image.

grayscale - See *greyscale.*

greyscale - When video is displayed in shades of black and white.

grouping - For Media Control Interface devices, refers to the ability to associate dissimilar devices for a common purpose. Grouping MCI devices aids resource management by insuring that all devices in a group are kept together.

H

handshaking - The exchange of predetermined signals when a connection is established between two data set devices.

help view - A view of an object that provides information to assist users in working with that object.

Hi8 - High-band 8mm videotape format.

HID - Handler identification.

High Sierra Group (HSG) - (1) A group that set the standards for information exchange for a CD-ROM. (2) HSG also refers to those standards (HSG standards).

HMS - (1) Hours-minutes-seconds. (2) A time format for videodisc players.

HMSF - (1) Hours-minutes-seconds-frames. (2) A time format for videodisc players.

hot spot - The area of a display screen that is activated to accept user input. Synonym for *touch area* .

HSG - High Sierra Group.

HSI - Hue saturation intensity.

hue - Describes the position of a color in a range from blue to green. A hue level of 0 is maximum blue. A hue level of 100 is maximum green. Synonym for *tint* .

hue saturation intensity (HSI) - A method of describing color in three-dimensional color space.

HWID - Hardware identifier.

hypermedia - Navigation or data transfer between connected objects of different media types. For example, a user might navigate from an image object to an audio object that describes the image over a hypermedia link. Or, a representation of a graph object might be embedded in a document object with a hypermedia data transfer link, such that when the graph object is changed, the representation of that object in the document is also changed.

Hytime - An ANSI standard proposal that addresses the synchronization and linking of multimedia objects.

I

IDC - Inter-device communication mechanism provided by the OS/2 function ATTACHDD DevHelp.

identifier - (1) A sequence of bits or characters that identifies a program, device, or system to another program, device, or system. (2) In the C language, a sequence of letters, digits, and underscores used to identify a data object or function. (3) See *four-character code (FOURCC)* .

IDOCA - Integrated Data Object Content Architecture.

image - An electronic representation of a video still.

image bitsperpel - Pertaining to the number of colors supported by the current pel format. The currently accepted standard values are those supported by OS/2 bitmaps, for example, 1, 4, 8, or 24 bits per pel. In addition, 12 bits per pel formats are accepted for YUV images (including the 'yuvb' pel format for RDIB files).

image buffer - A location in memory where video images are stored for later use.

image buffer formats - The format or representation of data buffers containing video images.

image compression - The method of compressing video image data to conserve storage space.

image file format - The format or representation of data files containing video images.

Image Object Content Architecture (IOCA) - A data format for multimedia products.

image pelformat - Indicates the color representation that is to be used for images that are captured and saved. This normally includes palettized RGB, true-color RGB, or YUV color formats.

image quality - Represents the user's or application's subjective evaluation of complexity and quality of the image to be captured or saved. This setting is used to determine specific compression methods to use for saving the image.

implicit event - An event that all stream handlers always must support, such as *end of stream* or *preroll complete*).

in-betweening - Synonym for *tweening* .

in frame - Refers to a subject that is included within the frame. See also *frame* .

InfoWindow system - A display system that can combine text, graphics, and video images on a single display. The minimum system configuration is the IBM InfoWindow Color Display, a system unit, a keyboard, and one or two videodisc players.

input locking mask - A filter, or mask, that controls which areas of the display can display or freeze video.

input/output control (IOCtl) - A system function that provides a method for an application to send device-specific control commands to a device driver.

installable I/O procedure - A file format handler that provides functions that operate on the media object of a particular data format. These functions include opening, reading, writing, seeking, and closing elements.

instance - See *node instance* . (RTMIDI-specific term)

INT - Script abbreviation for *interior* .

Integrated Data Object Control Architecture (IDOCA) - A data format for multimedia products.

interactive multimedia - The delivery of information content through combinations of video, computer graphics, sound, and text in a manner that allows the user to interact.

interactive program - A running program that can receive input from the keyboard or another input device. Contrast with *noninteractive program* .

interactive videodisc system (IVS) - A system in which a user can interact with a videodisc display image by entering commands to the computer through a device such as a keyboard or keypad or by touching a touch-sensitive screen at specific points on the display surface.

interlace flicker - The apparent flicker when one field of an interlaced image contains more light than the other field due to the placement of image details with respect to the separate fields. Two methods used to avoid interlace flicker: limit the vertical resolution on natural images (as opposed to text or graphics); design characters so that each character has an equal number of pels in each field.

interlacing - In multimedia applications, a characteristic of video image display that results in greater image clarity. In effect, the video image is traced across the screen twice. (The time delay between the two tracings makes this effect undesirable for normal computer-generated graphics.) Synonymous with *interleaving* .

interleaving - (1) The simultaneous accessing of two or more bytes or streams of data from distinct storage units. (2) The alternating of two or more operations or functions through the overlapped use of a computer facility. (3) In a duplicator, the process of inserting absorbent sheets between successive sheets of the copy paper to prevent set-off. (T) (4) Synonym for interlacing.

internal I/O procedure - An I/O procedure that is built in to the MMPM/2 system, including DOS, MEM, BND, and CF IOProcs.

International Organization for Standardization (ISO) - An organization of national standards bodies from various countries established to promote development of standards to facilitate international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

IOCtl - Input/output control.

IOProc - A file format handler that provides functions that operate on the media object of a particular data format. These processes include opening, reading, writing, seeking, and closing elements of a storage system. There are two classes of I/O procedures: *file format* and *storage system* .

iris - To fade a picture by operating the iris (aperture) on the camera in a certain way; the type of computer image dissolve accomplished by operating the aperture in a certain way.

ISO - International Organization for Standardization.

ISP - IBM Signal Processor. An IBM proprietary digital signal processor.

ISPOS - IBM Signal Processor Operating System.

ISV - Independent software vendor.

items - Options, choices or keywords such as one or more of the following options, choices or keywords can or should be specified. Sometimes use of these items can also be exclusive or some items may not be compatible with other items.

IVS - Interactive videodisc system.

J

JIT - Just in time. (Often used with "learning" as "JIT learning".) Multimedia help functions, closely integrated with applications, that employ voice output to guide the user.

Joint Photographic Experts Group (JPEG) - A group that is working to establish a standard for compressing and storing still images in digital form. JPEG also refers to the standard under development by this group (JPEG standard).

joy stick - In computer graphics, a lever that can pivot in all directions and that is used as a locater device. (Resembles an airplane's joy stick).

JPEG - Joint Photographic Experts Group.

K

keeper - Synonym for *buy* .

kernel - (1) The part of an operating system that performs basic functions such as allocating hardware resources. (2) A program that can run under different operating system environments. (3) A part of a program that must be in main storage in order to load other parts of the program.

key frames - The start and end frames of a single movement in an animation sequence; also can refer to the periodic full-frame image interspersed in the stream to allow random starts from these full-frame images (key frames).

keypad - A small, often hand-held, keyboard.

L

laser - Light amplification by stimulated emission of radiation; the device that produces this light.

latency - In video, the time it takes for the light from the phosphor screen to decay after the excitation is removed. Long-persistence phosphor has less flicker of still images, but more blurring of moving images.

level one videodisc applications - Interactive applications based on manual keypad functions, picture stops, and chapter stops.

level three videodisc applications - Interactive applications controlled by an external computer that uses the videodisc player as a peripheral device.

level two videodisc applications - Interactive applications controlled by the keypad and the videodisc player's internal computer. The control program is recorded on the videodisc itself.

LIB - Dynamic-link definition library. A file containing the data needed to build a program .EXE file but which does not contain the dynamic-link programs themselves. Contrast with *dynamic-link library* .

light pen - A light-sensitive pick device that is used by pointing it at the display surface.

line graphics - Synonym for *coordinate graphics* .

line pairing - A faulty interlace pattern in which the lines of the second field begin to pair with the lines of the first field rather than fit exactly within them.

linear audio - The analog audio on the linear track of videotape that can be recorded without erasing existing video; used for audio dubbing after video is edited.

linear video - A sequence of motion footage played from start to finish without stops or branching, like a movie.

link - A one-way link (directed edge) from one instance to another. If an instance wishes to send a message, it is sent along all the links from it. An instance can have any number of links coming from it, and any number of other instances can have links to it. An instance cannot select along which links the message should be sent. See also *slot* . (RTMIDI-specific term)

LIST chunk - A chunk that contains a list or an ordered sequence of subchunks.

list type - (1) A field in the first four bytes of the data field of a LIST chunk. (2) A four-character code identifying the contents of the list.

locked memory - An area of memory that is not available for use because it is being held by another process.

long shot - (1) A camera angle that reveals the subject and the surroundings. Often used as an establishing shot. (2) Synonym for *wide shot* .

LS - Script abbreviation for *long shot* .

luminance signal - The portion of image information that provides brightness. Alone, luminance provides a monochrome image.

M

M-ACPA - M-Audio Capture and Playback Adapter.

M-Audio Capture and Playback Adapter (M-ACPA) - An adapter card (for use with the IBM PS/2 product line) that provides the ability to record and play back high quality sound. The adapter converts the audio input (analog) signals to a digital format that is compressed and stored for later use.

master stream handler - Controls the behavior of one or more subordinate objects (the *slave streams*).

matte - In film, an opaque piece of art or a model that leaves a selected area unexposed to be filled on a subsequent pass or in composite.

MCD - Media control driver.

MCI - Media Control Interface.

M-Control Program/2 - The software interface required for the M-Motion Video Adapter/A. It consists of APIs or toolkits for DOS, Windows, Windows MCI, OS/2, and OS/2 MMPM/2. It also includes Pioneer and Sony videodisc player drivers for these environments. See also *M-Motion Video Adapter/A* .

MDM - Media device manager.

media - More than one hardware medium.

media component - A processor of audiovisual information or media. Media components can be either internal or external physical devices or defined mechanisms for effecting higher-level function from internal hardware and software subsystems. (An example is a waveform player component that utilizes the DSP subsystem and data streaming services to effect audio playback functions.)

media component capabilities - The functionality of a media component, including component functions that are supported.

media component type - A class of media components that exhibit similar behavior and capabilities. Examples of media component types are analog video display hardware and MIDI synthesizers.

media control driver (MCD) - A software implementation or method that effects the function of a media component. For OS/2, a media control driver or *media driver* is a dynamic-link library (or set of libraries) that utilizes physical device drivers, Media Device Manager services, and OS/2 to implement the function of the media component.

Media Control Interface (MCI) - A generalized interface to control multimedia devices. Each device has its own MCI driver that implements a standard set of MCI functions. In addition, each media driver can implement functions that are specific to the particular device.

media device - A processor of audiovisual information or media. Media components can be either internal or external physical devices or defined mechanisms for effecting higher-level function from internal hardware and software subsystems. (An example is a waveform player component that utilizes the DSP (Digital Signal Processor) subsystem and data-streaming services to effect audio-playback functions.)

media device capabilities - The functionality of a media component, including supported component functions.

media device connection - A physical or logical link between media component connectors for a particular set of media component instances.

media device connector - A physical or logical input or output on a media component.

media device connector index - An identifier for a media component connector.

media device ID - Media component identification. A unique identifier for a component.

media device instance - A case of an application's use of a media component.

media device manager (MDM) - A system service that, when two or more applications attempt to control a media device, determines which process gains access.

media driver - A device driver for a multimedia device. See also *device driver* .

media driver - A software implementation or method that effects the function of a media device.

media element manager (MEM) - A system service that manipulates multimedia data.

media programming interface (MPI) - A subsystem that provides a comprehensive system programming API layer for multimedia applications.

media segment - An audiovisual object of some type, such as a waveform, song, video clip, and so on.

media unit - A medium on which files are stored; for example, a diskette.

media volume - A (possibly heterogeneous) physical or logical collection of media segments. (Examples are a videodisc, video tape, compact audio disc, OFF file, and RIFF file.)

media volume file - A media volume that is embodied as a conventional binary computer file within a computer file system on storage devices such as disks, diskettes, or CD-ROMs. Such storage devices can be either local or remote. Media volume files can be of various formats, such as OFF or RIFF, and contain segments of various types.

medium shot - A camera angle that reveals more of the subject than a close-up but less than a wide shot, usually from face to waistline; sometimes called a *mid-shot* .

MEM - Media element manager.

MEM IOProc - An internal I/O procedure provided by the MMPM/2 system that supports memory files.

memory file - A block of memory that is perceived as a file by an application.

memory playlist - A data structure in the application used to specify the memory addresses to play from or record to. The application can modify the playlist to achieve various effects in controlling the memory stream.

message interface - See *command message interface* .

MIDI Mapper - Provides the ability to translate and redirect MIDI messages to achieve device-independent playback of MIDI sequences.

MIDI message - A sequence of bytes that conform to the MIDI standard. There are two categories: System Exclusive (SysEx) and non-SysEx messages. SysEx messages can be of any length, whereas non-SysEx messages are between one and three bytes.

mid-shot - See *medium shot* .

millisecond - One thousandth of a second.

minutes-seconds-frames (MSF) - A time format based on the 75-frames-per-second CD digital audio standard.

MIPS - Millions of instructions per second. A unit of measure of processing performance equal to one million instructions per second.

mix - The combination of audio or video sources during postproduction.

mixed-media system - Synonym for *multimedia system* .

Mixed Object : Document Content Architecture (MO:DCA) - A data format for multimedia products.

mixer - A device used to simultaneously combine and blend several inputs into one or two outputs.

mixing - (1) In computer graphics, the result of the intersection of two or more colors. (2) In filming, the combining of audio and video sources that is accomplished during postproduction at the *mix* . (3) In recording, the combining of audio sources.

MMIO - Multimedia input/output.

MMIO file services - System services that enable an application to access and manipulate multimedia data files.

MMIO manager - Multimedia input/output manager. The MMIO manager provides services to find, query, and access multimedia data objects. It also supports the functions of memory allocation and file compaction. The MMIO manager uses IOProcs to direct the input and output associated with reading from and writing to different types of storage systems or file formats.

M-Motion - A multimedia platform that offers analog video in addition to quality sound and images. The M-Motion environment consists of the M-Motion Video Adapter/A and the M-Control Program/2 master stream handler.

M-Motion Video Adapter/A - (1) A Micro Channel adapter that receives and processes signals from multiple video and audio sources, and then sends these signals to a monitor and speakers. (2) Dual plane video hardware that offers analog video in addition to quality sound and images. (3) The M-Motion Video Adapter/A requires the M-Control Program/2.

MMPM/2 - Multimedia Presentation Manager/2. See *OS/2 multimedia* .

MMTIME - Standard time and media position format supported by the media control interface. This time unit is 1/3000 second, or 333 microseconds.

MO:DCA - Mixed Object : Document Content Architecture.

mode - A method of operation in which the actions that are available to a user are determined by the state of the system.

model - The conceptual and operational understanding that a person has about something.

module - A language construct that consists of procedures or data declarations and that can interact with other constructs.

moire - An independent, usually shimmering pattern seen when two geometrically regular patterns (as a sampling frequency and a correct frequency) are superimposed. The moire pattern is an alias frequency. See also *aliasing* .

monitor - See *video monitor* .

monitor window - A graphical window, available from a digital video device, which displays the source rectangle, and any subset of this video capture region. See *destination rectangle* for related information.

motion-control photography - A system for using computers to precisely control camera movements so that the different elements of a shot-models and backgrounds, for example-can later be composited with a natural and believable unity.

motion video capture adapter - An adapter that, when attached to a computer, allows an ordinary television picture to be displayed on all or part of the screen, mixing high-resolution computer graphics with video; also enables a video camera to become an input device.

Motion Video Object Content Architecture (MVOCA) - A data format for multimedia products.

Moving Pictures Experts Group (MPEG) - A group that is working to establish a standard for compressing and storing motion video and animation in digital form.

MPEG - Moving Pictures Experts Group.

MPI - Media Programming Interface.

MPI application services - Media Programming Interface application services. Functional services provided by MPI to application programs and higher-level programming constructs, such as multimedia controls.

MS - Script abbreviation for *medium shot* .

MSF - Minutes-seconds-frames.

multimedia - Material presented in a combination of text, graphics, video, image, animation, and sound.

multimedia data object - In an application, an element of a data structure (such as a file, an array, or an operand) that is needed for program execution and that is named or otherwise specified by the allowable character set of the language in which the program is coded.

Multimedia File I/O Services - System services that provide a generalized interface to manipulate multimedia data. The services support buffered and unbuffered file I/O, standard RIFF files, and installable I/O procedures.

multimedia input/output (MMIO) - (1) System services that provide a variety of functions for media file access and manipulation. (2) A consistent programming interface where an application, media driver, or stream handler can refer to multimedia files, read and write data to the files, and query the contents of the files, while remaining independent of the underlying file formats or the storage systems that contain the files.

multimedia navigation system - A tool that gives the information product designer the freedom to link various kinds and pieces of data in a variety of ways so that users can move through it nonsequentially.

multimedia system - (1) A system capable of presenting multimedia material in its entirety. (2) Synonym for *mixed-media system* .

multiple selection - A selection technique in which a user can select any number of objects, or not select any.

Musical Instrument Digital Interface (MIDI) - A protocol that allows a synthesizer to send signals to another synthesizer or to a computer, or a computer to a musical instrument, or a computer to another computer.

mute - To temporarily turn off the audio for the associated medium.

mux - An abbreviation for multiplexer. See also *mixer* .

MVOCA - Motion Video Object Content Architecture.

N

NAPLPS - North American Presentation Level Protocol Syntax.

National Television Standard Committee (NTSC) - A committee that set the standard for color television broadcasting and video in the United States (currently in use also in Japan); also refers to the standard set by this committee (NTSC standard).

node - An abstract term indicating either a node class or a node instance. When used with a class qualifier (for example, application node) it implies an instance (for example, instance of an application class). (RTMIDI-specific term)

node class - A definition of the behavior of a node instance. All instances of the same class are expected to have the same behavior and purpose, although this restriction is not enforced by the driver. (RTMIDI-specific term)

node instance - A vertex in the node network that can receive and transmit MIDI messages. (RTMIDI-specific term)

node network - The collection (graph) of node instances and links. (RTMIDI-specific term)

nondrop time code - Synonym for *full-frame time code* .

noninteractive program - A running program that cannot receive input from the keyboard or other input device.

non-streaming device - (1) A device that contains both source and destination information for multimedia. (2) A device that transmits data (usually analog) directly, without streaming to system memory.

North American Presentation Level Protocol Syntax (NAPLPS) - A protocol used for display and communication of text and graphics in a videotex system; a form of vector graphics.

notebook - A graphical representation that resembles a perfect-bound or spiral-bound notebook that contains pages separated into sections by tabbed divider pages. A user can turn the pages of a notebook to move from one section to another.

NTSC - National Television Standard Committee.

null streaming - (1) The behavior of a stream that can be created and started but which has no associated data flow. (2) The behavior of a stream that can be created and started but which has no associated data flow. For example, a CD-DA is a non-streaming device. (3) A device that does not stream its data through the MMPM/2 streaming system For example, a CDDA device is a non-streaming device.

O

object - (1) Anything that exists in and occupies space in storage and on which operations can be performed; for example, programs, files, libraries, and folders. (2) Anything to which access is controlled; for example, a file, a program, an area of main storage. (3) See also *data object* and *media object* .

object-action paradigm - A method where users select the object that they want to work with, then choose the action they wish to perform on that object. See *object orientation* .

object class - A categorization or grouping of objects that share similar behaviors and characteristics.

object connection - A link between two objects. Connections can be used for navigation, as with *hypermedia,* or for data transfer between objects.

Object Content Architecture (OCA) - A data format for multimedia products.

object decomposition - The process of breaking an object into its component parts.

object orientation - An orientation in a user interface in which a user's attention is directed toward the objects the user works with, rather than applications, to perform a task.

object-oriented user interface - A type of user interface that implements object orientation and the object-action paradigm.

object template - An object that can be used to create another object of the same object class. The template is a basic framework of the object class, and the newly created object is an instance of the object class.

OCA - Object Content Architecture.

OEM - Original equipment manufacturer.

OFF - Operational file format.

offline edit - A preliminary or test edit usually done on a low-cost editing system using videocassette work tapes. (An offline edit is done so that decisions can be made and approvals given prior to the final edit.)

online edit - The final edit, using the master tapes to produce a finished program.

operational file format (OFF) - A file format standard.

optical disc - A disc with a plastic coating on which information (as sound or visual images) is recorded digitally as tiny pits and read using a laser. The three categories of optical discs are CD-ROM, WORM, and erasable.

optical drive - Drives that run optical discs.

optical reflective disc - A designation of the means by which the laser beam reads data on an optical videodisc. In the case of a reflective disc, the laser beam is reflected off a shiny surface on the disc.

opticals - Visual effects produced optically by means of a device (an optical printer) that contains one camera head and several projectors. The projectors are precisely aligned so as to produce multiple exposures in exact registration on the film as in the camera head.

original footage - The footage from which the program is constructed.

OS/2 multimedia - A subsystem service of OS/2 that provides a software platform for multimedia applications. It defines standard interfaces between multimedia devices and OS/2 multimedia applications.

overlay - The ability to superimpose text and graphics over video.

overlay device - Provides support for video overlaying along with video attribute elements. The video overlaying handles tasks such as displaying, and sizing video. Synonym for *video overlay device* .

P

PAL - Phase Alternation Line.

palette - See *color palette* , *standard palette* , and *custom palette* .

pan - A camera movement where the camera moves sideways on its stationary tripod; left-to-right balance in an audio system.

panel - A particular arrangement of information grouped together for presentation to users in a window.

panning - Progressively translating an entire display image to give the visual impression of lateral movement of the image.

In computer graphics, the viewing of an image that is too large to fit on a single screen by moving from one part of the image to another.

paradigm - An example, pattern, or model.

patch mapping - The reassignment of an instrument patch number associated with a specific synthesizer to the corresponding standard patch number in the General MIDI specification.

pause - To temporarily halt the medium. The halted visual should remain displayed but no audio should be played.

pause stop - In data streaming, a stop that pauses the data stream but does not disturb any data.

PDC - Physical device component.

PDD - Physical device driver.

pedestal up/down - A camera movement where the camera glides up or down on a boom.

pel - The dimensions of a toned area at a picture element. See also *picture element* .

Phase Alternation Line (PAL) - Television broadcast standard for European video outside of France and the Soviet Union.

physical device driver (PDD) - A program that handles hardware interrupts and supports a set of input and output functions.

picon - A graphic or natural image reduced to icon size. Similar to *thumbnail.*

picture element - In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity. See also *pel* .

picture-in-picture - A video window within a larger video window.

pixel - See *picture element* .

plaintext - Nonencrypted data.

platform - In computer technology, the principles on which an operating system is based.

play backward - To play the medium in the backward direction.

playback window - The graphic window in which software motion video is displayed. This window can be supplied by an application, or a default window can be created by a digital video device.

play forward - To play the medium in the forward direction.

pointer - A symbol, usually in the shape of an arrow, that a user can move with a pointing device. Users place the pointer over objects they want to work with.

pointing device - A device, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

polish - The version of the script submitted for final approval.

polyphony - A synthesizer mode where more than 1 note can be played at a time. Most synthesizers are 16-note to 32-note polyphonic.

postproduction - The online and offline editing process.

PPQN - (1) Parts-per-quarter-note. (2) A time format used in musical instrument digital interface (MIDI).

preproduction - The preparation stage for video production, when all logistics are planned and prepared.

preroll - The process of preparing a device to begin a playback or recording function with minimal latency. During a multimedia sequence, it might require that two devices be cued (prerolled) to start playing and recording at the same time.

primary window - A window in which the main interaction between a user and an object takes place.

Proc - A custom procedure, called by the particular utility manager, to handle input or output to files of a format different from DOS, MEM, or BND; for example, AVC or TIFF. By installing custom procedures, existing applications no longer need to store multiple copies of the same media file for running on various platforms using different file formats. See also *static resource* and *dynamic resource* .

production - In videotaping, the actual shooting.

production control room - The room or location where the monitoring and switching equipment is placed for the direction and control of a television production.

progress indicator - A control, usually a read-only slider, that informs the user about the status of a user request.

props - In videotaping, support material for the shoot, for example, equipment being promoted, auxiliary equipment, software, or supplies; anything provided to make the set look realistic and attractive.

protection master - A copy of the edit master that is stored as a backup.

PS/2 CD-ROM-II Drive - An IBM CD-ROM drive that can play compact disc digital audio (CD-DA) and CD-ROM/XA interleaved audio, video, and text, and adheres to the Small Computer System Interface (SCSI). The drive can be installed on Micro Channel and non-Micro Channel IBM PS/2 systems.

pulse code modulation (PCM) - In data communication, variation of a digital signal to represent information.

push button - A graphical control, labeled with text, graphics, or both, that represents an action that will be initiated when a user selects it. For example, when a user clicks on a *Play* button, a media object begins playing.

R

raster graphics - Computer graphics in which a display image is composed of an array of pels arranged in rows and columns.

raw footage - Synonym for *original footage* .

ray-tracing - A technique used by 3-D rendering software programs that automatically figures an object's position in three dimensions and calculates shadows, reflections, and hidden surfaces based on user-entered light locations and material characteristics. (In other words, if the user orders an object to be a mirror, the computer produces the mirror with all its correct reflective properties.)

real time - (1) Pertaining to the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process. This term is also used to describe systems operating in conversational mode and processes that can be influenced by human intervention while they are in progress. (2) A process control system or a computer-assisted instruction program, in which response to input is fast enough to affect subsequent output.

real-time recording - Refers to the capturing of video and audio data in real time, as the analog signals are generated from the video source device. The video source device can be a camcorder, or a videotape or videodisc player.

record - To transfer data from one source (for example, microphone, CD, videodisc) or set of sources to another medium.

Redbook audio - The storage format of standard audio CDs. See also *compact disc, digital audio (CD-DA)* .

reference frame - (1) Refers to the complete frame that is created at periodic intervals in the output stream. An editing operation always begins at a reference frame. (2) Synonymous with *key frame* and *I-frame* . (3) See *delta frame* .

reflective disc - See *optical reflective disc* .

render - In videotaping, to create a realistic image from objects and light data in a scene.

repeat - A mode which causes the medium to go to the beginning and start replaying when it reaches the medium's end.

resolution - (1) In computer graphics, a measure of the sharpness of an image, expressed as the number of lines and columns on the display screen or the number of pels per unit of area. (2) The number of lines in an image that an imaging system (for example, a telescope, the human eye, a camera, and so on) can resolve. A higher resolution makes text and graphics appear clearer.

resource - As used in the multimedia operating system, any specific unit of data created or used by a multimedia program. See also *static resource* and *dynamic resource* .

resource handler - A system service that loads, saves, and manipulates multimedia program units of data.

resource interchange file format (RIFF) - A tagged file format framework intended to be the basis for defining new file formats.

resync - Recovery processing performed by sync-point services when the failure of a session, transaction program, or LU occurs during sync-point processing. The purpose of resync is to return protected resources to consistent states.

resync tolerance value - A minimum time difference expressed in MMTIME format.

Revisable Form Text : Document Content Architecture (RFT:DCA) - A data format for multimedia products.

rewind - To advance the medium in the backward direction quickly, and optionally allow the user to scan the medium.

RFT:DCA - Revisable Form Text : Document Content Architecture.

RGB - Color coding where the brightness of the additive primary colors of light, red, green, and blue, are specified as three distinct values of white light.

RIFF - Resource interchange file format.

RIFF chunk - A chunk with a chunk ID of *RIFF* . In a RIFF file, this must be the first chunk.

RIFF compound file - A file containing multiple file elements or one file element that makes up the entire RIFF file. The MMIO manager provides services to find, query, and access any file elements in a RIFF compound file. Synonym for *bundle file* .

rotoscope - A camera setup that projects live-action film, one frame at a time, onto a surface so that an animator can trace complicated movements. When filmed, the completed animation matches the motion of the original action.

rough cut - (1) The result of the offline edit. (2) A video program that includes the appropriate footage in the correct order but does not include special effects.

RTMIDI - Real-time MIDI subsystem.

S

SAA - Systems Application Architecture.

safety - An extra shot of a scene that is taken as a backup after an acceptable shot (the *buy*) has been acquired.

saturation - The amounts of color and grayness in a hue that affect its vividness; that is, a hue with high saturation contains more color and less gray than a hue with low saturation. See also *hue* .

sampler - A device that converts real sound into digital information for storage on a computer.

scan backward - To display the video and optionally play the audio while the medium is advancing in the backward direction rapidly.

scan converter - A device that converts digital signal to NTSC or PAL format.

scan forward - To display the video and optionally play the audio while the medium is advancing in the forward direction rapidly.

scan line - (1) In a laser printer, one horizontal sweep of the laser beam across the photoconductor. (2) A single row of picture elements.

scanner - A device that examines a spatial pattern, one part after another, and generates analog or digital signals corresponding to the pattern.

SCB - Subsystem control block.

scene - A portion of video captured by the camera in one continuous shot. The scene is shot repeatedly (each attempt is called a *take*) until an acceptable version, called the *buy* , is taken.

scheduler - The code responsible for passing messages to instances. The scheduler has two queues, one for normal real-time MIDI messages (Q1), and the other for low-priority SysEx messages (Q2). (RTMIDI-specific term)

scheduler daemon - A small executable program, MIDIDMON.EXE, which is used to provide deferred-interrupt processing for the scheduler. The daemon is a super high-priority thread which gets blocked in ring 0. When the scheduler needs to run, this thread is unblocked. When the scheduler is finished, it re-blocks itself. When the unblocking occurs during an interrupt, the OS/2 kernel runs the daemon thread immediately after the interrupt handler has exited. This approach guarantees that the scheduler runs at task time.

scripting - Writing needed dialog.

scroll bar - A window component that shows a user that more information is available in a particular direction and can be scrolled into view. Scroll bars should not be used to represent an analog setting, like volume. Sliders should be used.

SCSI - Small computer system interface.

SECAM - Sequential Couleurs a Memoire. The French standard for color television.

secondary window - A window that contains information that is dependent on information in a primary window and is used to supplement the interaction in the primary window.

secondary window manager - A sizable dialog manager that enables application writers to use CUA-defined secondary windows instead of dialog boxes.

second generation - A direct copy from the master or original tape.

selection - The act of explicitly identifying one or more objects to which a subsequent choice will apply.

selection technique - The method by which users indicate objects on the interface that they want to work with.

semaphore - (1) A variable that is used to enforce mutual exclusion. (T) (2) An indicator used to control access to a file; for example, in a multiuser application, a flag that prevents simultaneous access to a file.

sequencer - A digital tape recorder.

set - In videotaping, the basic background or area for production.

settings - Characteristics of objects that can be viewed and sometimes altered by the user. Examples of a file's settings include name, size, and creation date. Examples of video clip's settings include brightness, contrast, color, and tint.

settings view - A view of an object that provides a way to change characteristics and options associated with the object.

SFX - Script abbreviation for *special effects* .

shade - To darken with, or as if with, a shadow; to add shading to.

sharpness - Refers to the clarity and detail of a video image. A sharpness value of 0 causes the video to be generally fuzzy with little detail. A sharpness value of 100 causes the video to be generally very detailed, and may appear grainy.

SHC - Stream handler command.

shoot - To videotape the needed pictures for the production.

shooting script - Synonym for *final script* .

shot list - A list containing each shot needed to complete a production, usually broken down into a schedule.

simple device - A multimedia device model for hardware which does not require any additional objects, known as device elements, to perform multimedia functions.

sine wave - A waveform that represents periodic oscillations of a pure frequency.

single plane video system - Refers to when video and graphics are combined into one buffer. This may appear the same as a dual plane video system, but since all the data is in one buffer, capture and restore operations will obtain both graphics and video components in one operation. See also *dual plane video system* .

single selection - A selection technique in which a user selects one, and only one, item at a time.

slave stream - A stream that is dependent on the *master stream* to maintain synchronization.

slave stream handler - In SPI, regularly updates the sync pulse EVCB with the stream time. The Sync/Stream Manager checks the slave stream handler time against the master stream time to determine whether to send a sync pulse to the slave stream handler.

slider - A visual component of a user interface that represents a quantity and its relationship to the range of possible values for that quantity. A user can also change the value of the quantity. Sliders are used for volume and time control.

slider arm - The visual indicator in the slider that a user can move to change the numerical value.

slider button - A button on a slider that a user clicks on to move the slider arm one increment in a particular direction, as indicated by the directional arrow on the button.

slide-show presentation - Synonym for *storyboard* .

slot - A distinct position in an instance from which links can be attached. The same message is sent along all links on a slot, but an instance can determine at run-time on which slots the message should be sent. An instance can support multiple slots if it wants to be able to send different messages to different targets. (RTMIDI-specific term)

small computer system interface (SCSI) - An input and output bus that provides a standard interface between the OS/2* multimedia system and peripheral devices.

SMH - Stream manager helper.

SMPTE - Society of Motion Picture and Television Engineers.

SMPTE time code - A frame-numbering system developed by SMPTE that assigns a number to each frame of video. The 8-digit code is in the form HH:MM:SS:FF (hours, minutes, seconds, frame number). The numbers track elapsed hours, minutes, seconds, and frames from any chosen point.

SMV - Software motion video.

socketable user interface - An interface defined by multimedia controls that enable the interface to be plugged into and unplugged from applications without affecting the underlying object control subsystem.

sound track - Synonym for *audio track* .

source node - An instance which can generate a compound message. Hardware nodes generate messages from data received from Type A drivers. Application nodes generate them from data sent from an application. (RTMIDI-specific term)

source rectangle - An abstract region representing the area available for use by a video capture adapter. This window is displayed in the monitor window of the digital video device. A subset of the maximum possible region to be captured can be defined; such a subset is shown by an animated dashed rectangle in the monitor window.

source window - See *source rectangle* .

SPCB - Stream protocol control block.

special effects - In videotaping, any activity that is not live footage, such as digital effects, computer manipulation of the picture, and nonbackground music.

SPI - Stream programming interface.

split streaming - A mechanism provided by the Sync/Stream Manager to create one data stream source with multiple targets.

SPP - A time format based on the number of beats-per-minute in the MIDI file.

sprite - An animated object that moves around the screen without affecting the background.

sprite graphics - A small graphics picture, or series of pictures, that can be moved independently around the screen, producing animated effects.

squeeze-zoom - A DVE where one picture is reduced in size and displayed with a full-screen picture.

SSM - Sync/Stream Manager.

standard multimedia device controls - These controls provide the application developer with a CUA compliant interface for controlling audio attributes, video attributes, and videodisc players. These controls simplify the programming task required to create the interface and handle the presentation of the interface and all interaction with the user. They also send the Media Control Interface (MCI) commands to the Media Device Manager (MDM) for processing.

standard objects - A set of common, cross-product objects provided and supported by the system. Examples include folders, printers, shredders, and media players.

standard palette - A set of colors that is common between applications or images. See also *custom palette* and *color palette* .

static resource - A *resource* that resides on any read-and-write or read-only medium. Contrast with *dynamic resource* .

status area - Provides information as to the state of the medium and device, or both. It should indicate what button is currently pressed and what modes (for example, mute) are active.

step backward - To move the medium backward one frame or segment at a time.

step forward - To move the medium forward one frame or segment at a time.

still - A static photograph.

still image - See *video image* .

still video capture adapter - An adapter that, when attached to a computer, enables a video camera to become an input device. See also *motion video capture adapter* .

stop - Halt (stops) the medium.

storage system - The method or format a functional unit uses to retain or retrieve data placed within the unit.

storage system IOProc - A procedure that unwraps data objects such as RIFF files, RIFF compound files, and AVC files. IOProcs are ignorant of the content of the data they contain. A storage system IOProc goes directly to the OS/2 file system (or to memory in the case of a MEM file) and does not pass information to any other file format or storage system IOProc. The internal I/O procedures provided for DOS files, memory files, and RIFF compound files are examples of storage system IOProcs, because they operate on the storage mechanism rather than on the data itself. See also *file format IOProc* .

storyboard - (1) A visual representation of the script, showing a picture of each scene and describing its corresponding audio. (2) Synonym for *slide-show presentation* .

storyboarding - Producing a sequence of still images, such as titles, graphics, and images, to work out the visual details of a script.

stream - To send data from source to destination via buffered system memory.

stream connector - A port or connector that a device uses to send or receive. See also *connector* .

stream handler - A routine that controls a program's reaction to a specific external event through a continuous string of individual data values.

stream handler command (SHC) - Synchronous calls provided by both ring 3 DLL stream handlers as a DLL call and by ring 0 PDD stream handlers as a IDC call. The stream handler commands are provided through a single entry point, SHCEntryPoint, which accepts a parameter structure on input. This enables the DLL and PDD interfaces to the stream manager to be the same.

stream manager - A system service that controls the registration and activities of all stream handlers.

stream manager helper (SMH) - Routines provided by the stream manager for use by all stream handlers. The stream handlers use these helper routines to register with the manager, report events, and synchronize cues to the manager to request or return buffers to the manager. They are synchronous functions and are available to both ring 3 DLL stream handlers as a DLL call and to ring 0 PDD stream handlers.

stream programming interface - A system service that supports continual flow of data between physical devices.

stream programming interface (SPI) - A system service that supports continual flow of data between physical devices.

stream protocol control block (SPCB) - The system service that controls the behavior of a specified stream type. This enables you to subclass a stream's data type, change data buffering characteristics, and alter synchronization behavior and other stream events.

strike - In videotaping, to clear away, remove, or dismantle anything on the set.

subchunk - The first *chunk* in a RIFF file is a *RIFF chunk* ; all other chunks in the file are subchunks of the RIFF chunk.

subclassing - The act of intercepting messages and passing them on to their original intended recipient.

super - Titles or graphics overlaid on the picture electronically. See also *superimpose* .

superimpose - To overlay titles or graphics on the picture electronically.

S-video - (1) Separated video or super video. (2) A signal system using a Y/C format. (3) See also *Y/C* , *composite video* , and *component video* .

S-Video input connector - A special connector that separates the chrominance from the luminance signal.

sweetening - (1) The equalization of audio to eliminate noise and obtain the cleanest and most level sound possible. (2) The addition of laughter to an audio track.

switching - Electronically designating, from between two or more video sources, which source's pictures are recorded on tape. Switching can occur during a shoot or during an edit.

symmetric video compression - A technology in which the computer can be used to create, as well as play back, full-motion, full-color video.

sync - Synchronization or synchronized.

synchronization - The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

synchronous - Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals.

sync group - A *master stream* and all its *slaves* that can be started, stopped, and searched as a group by using the slaves flag on each of the following SPI functions:

- SpiStartStream
- SpiStopStream
- SpiSeekStream

sync pulse - A system service that enables each slave stream handler to adjust the activity of that stream so that synchronization can be maintained. Sync pulses are introduced by transmission equipment into the receiving equipment to keep the two equipments operating in step.

sync signal - Video signal used to synchronize video equipment.

synthesizer - A musical instrument that allows its user to produce and control electronically generated sounds.

system message - A predefined message sent by the MMIO manager for the message's associated function. For example, when an application calls mmioOpen, the MMIO manager sends an MMIOM_OPEN message to an I/O procedure to open the specified file.

Systems Application Architecture (SAA) - A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

T

tagged image file format (TIFF) - An easily transportable image file type used by a wide range of multimedia software.

take - During the shoot in videotaping, each separate attempt at shooting a scene. This is expressed as: Scene 1, Take 1; Scene 1, Take 2, and so on.

talent - On-screen person (professional or amateur) who appears before the camera or does voice-over narration.

TAM - Telephone answering machine.

target node - An instance which receives a message but does not forward it because it is the final instance in a chain of processing. For example, a hardware node is a target node because when it receives a message, it sends it to another device driver, and not to another instance. (RTMIDI-specific term)

tearing - Refers to when video is displaced horizontally. This may be caused by sync problems.

TelePrompTer - A special monitor mounted in front of a camera so that talent can read text and will appear to be looking at the camera.

thaw - See *unfreeze* .

thumbnail - A small representation of an object. For example, a full screen image might be presented in a much smaller area in an authoring system time line. A *picon* is an example of a thumbnail.

TIFF - Tagged Image File Format time code.

tilt - A camera movement where the camera pivots up or down on its stationary tripod.

timbre - The distinctive tone of a musical instrument or human voice that distinguishes it from other sounds.

time code - See *SMPTE time code* .

time-line processor - A type of authoring facility that displays an event as elements that represent time from the start of the event.

tint - See *hue.*

TMSF - A time format expressed in tracks, minutes, seconds, and frames, which is used primarily by compact disc audio devices.

tone (bass, treble, etc...) - A control that adjusts the various attributes of the audio.

tool palette - A palette containing choices that represent tools, often used in media editors (such as graphics and audio editors). For example, a user might select a "pencil" choice from the tool palette to draw a line in the window.

touch area - (1) An area of a display screen that is activated to accept user input. (2) Synonymous with *anchor, hot spot, and trigger* .

track - A path associated with a single Read/Write head as the data medium moves past it.

track advance - To advance the medium to the beginning of the next track.

track reverse - To rewind the medium to the beginning of the current track. If it is at the beginning of the track it will then jump to the beginning of the previous track.

transform device - A device that modifies a signal or stream received from a transport device. Examples are amplifier-mixer and overlay devices.

translator - A computer program that can translate. In telephone equipment the device that converts dialed digits into call-routine information.

transparency - Refers to when a selected color on a graphics screen is made transparent to allow the video "behind it" to become visible. Often found in dual plane video subsystems.

transparent color - Video information is considered as being present on the video plane which is maintained behind the graphics plane. When an area on the graphics plane is painted with a transparent color, the video information in the video plane is made visible. See also *dual plane video system* .

transport device - A device that plays, records, and positions a media element, and either presents the result directly or sends the material to a *transform device* . Examples are videodisc players, CD-ROMs, and digital audio (wave) player.

treatment - A detailed design document of the video.

tremolo - A vibrating effect of a musical instrument produced by small and rapid amplitude variations to produce special musical effects.

trigger - (1) An area of a display screen that is activated to accept user input. (2) Synonymous with *anchor, hot spot, and touch area* .

truck - In videotaping, a sideways camera movement of the tripod on which the camera is mounted.

tweening - (1) The process of having the computer draw intermediate animation frames between key frames. In other words, the animation tool requires only that pictures of key sections of a motion be provided; the software calculates all the in-between movements. (2) Synonym for *in-betweening* .

U

Ultimatte - The trade name of a very high-quality, special-effects system used for background replacement and image composites.

U-matic - A video cassette system using 0.75-inch tape format.

underrun - Loss of data caused by the inability of a transmitting device or channel to provide data to the communication control logic (SDLC or BSC/SS) at a rate fast enough for the attached data link or loop.

unfreeze - (1) To return to action after a *freeze* . (2) Enables updates to the video buffer. (3) Synonym for *thaw* .

unidirectional microphone - A microphone that responds to sound from only one direction and is not subject to change of direction. (A unidirectional microphone is the type of microphone employed in computers capable of voice recognition.)

unload - To eject the medium from the device.

user-defined message - A private message sent directly to an I/O procedure by using the mmioSendMessage function. All messages begin with an MMIOM prefix, with user-defined messages starting at MMIOM_USER or above.

user interface - The area at which a user and an object come together to interact. As applied to computers, the ensemble of hardware and software that allows a user to interact with a computer.

user's conceptual model - A user's mental model about how things should work. Much of the concepts and expectations that make up the model are derived from the user's experience with real-world objects of similar type, and experience with other computer systems.

V

value set - A control used to present a series of mutually exclusive graphical choices. A tool palette in a paint program can be implemented using a value set.

VCR - Videocassette recorder.

VDD - Virtual device driver.

VDH - Virtual device helper.

VDP - Video display processor.

vector graphics - See *coordinate graphics* .

vendor specific drivers - An extension to an MCD to execute hardware specific commands.

VHS - Very high speed. A consumer and industrial tape format (VHS format).

vicon - A vicon, or video icon, can be an animation or motion video segment in icon size. Usually this would be a short, repeating segment, such as an animation of a cassette tape with turning wheels.

video - Pertaining to the portion of recorded information that can be seen.

video aspect ratio - See *aspect ratio* .

video attribute control - Provides access to and operation of the standard video attributes: brightness, contrast, freeze, hue, saturation, and sharpness. All device communication and user interface support is handled by the control.

video attributes - Refers to the standard video attributes: brightness, contrast, freeze, hue, saturation, and sharpness.

video clip - A section of filmed or videotaped material.

video clipping - See *clipping* .

video digitizer - Any system for converting analog video material to digital representation. (For example, see *DVI* .)

video display buffer - The buffer containing the visual information to be displayed. This buffer is read by the video display controller.

video display controller - The graphics or video adapter that connects to a display and presents visual information.

video encoder - A device (adapter) that transforms the high-resolution digital image from the computer into a standard television signal, thereby allowing the computer to create graphics for use in video production.

video graphics adapter - A graphics controller for color displays. The pel resolution of the video graphics adapter is 4:4.

video image - (1) A still video image that has been captured. (2) Synonymous with *image* and *still image* .

video monitor - A display device capable of accepting a video signal that is not modulated for broadcast either on cable or over the air; in videotaping, a television screen located away from the set where the footage can be viewed as it is being recorded.

video overlay - See *overlay* .

video overlay device - See *overlay device* .

video plane - In a dual plane video system, the video plane contains the video. This video plane will be combined with the graphics plane to create an entire display image.

video programming interface (VPI) - A subsystem that performs output from video source to video window.

video quality - The compression quality level setting to be set for the CODEC. This value is in the range of 0 (min) - 100 (max).

video record rate - Frame rate for recording as an integral number of frames per second. This sets the target capture rate, but there are no assurances this rate will be attained. Drop frame records will be inserted into the output data stream to indicate frames dropped during the capture/record process.

video record frame duration - Frame rate for recording as the time duration of each frame in microseconds. Useful for setting non-integer frame rates, for example, 12.5 FPS of a PAL videodisc: 1000000/12.5 = 8000 microseconds.

video signal - An electrical signal containing video information. The signal must be in some standard format, such as NTSC or PAL.

VSD - Vendor Specific Driver

video scaling - (1) Expanding or reducing video information in size or area. (2) See also *aspect ratio* .

video scan converter - A device that emits a video signal in one standard into another device of different resolution or scan rate.

video segment - A contiguous set of recorded data from a video track. A video segment might or might not be associated with an audio segment.

video signal - An electrical signal containing video information. The signal must be in some standard format, such as NTSC or PAL.

video source selection - The ability of an application to change to an alternate video input using the **connector** command.

video tearing - See *tearing* .

video teleconferencing - A means of telecommunication characterized by audio and video transmission, usually involving several parties. Desktop video teleconferencing could involve having the audio and video processed by the user's computer system, that is, with the other users' voices coming through the computer's speaker, and video windows of the other users displayed on the computer's screen.

videocassette recorder (VCR) - A device for recording or playing back videocassettes.

videodisc - A disc on which programs have been recorded for playback on a computer (or a television set); a recording on a videodisc. The most common format in the United States and Japan is an NTSC signal recorded on the optical reflective format.

videodisc player control - Provides access to and operation of the following videodisc functions: eject, pause, play forward, play reverse, position, record, repeat, rewind, scan forward, scan reverse, step forward, step reverse, and stop. All device communication and user interface support is handled by the control.

videotape - (1) The tape used to record visual images and sound. (2) To make a videotape of. (3) A recording of visual images and sound made on magnetic tape. (All shooting is done in this format, even if the results are later transferred to videodisc or film.)

videotape recorder (VTR) - A device for recording and playing back videotapes. (The professional counterpart of a consumer VCR.)

videotex - A system that provides two-way interactive information services, including the exchange of alphanumeric and graphic information, over common carrier facilities to a mass consumer market using modified TV displays with special decoders and modems.

video windows - Graphical PM-style windows in which video is displayed. Most often associated with the video overlay device.

view - The form in which an object is presented. The four kinds of views are: composed, contents, settings, and help.

viewport - An area on the usable area of the display surface over which the developer has control of the size, location, and scaling, and in which the user can view all or a portion of the data outlined by the window.

virtual device helper - A system service that is available to perform essential functions.

VO - Script abbreviation for *voice-over* .

voice-over - (1) The voice of an unseen narrator in a video presentation. (2) A voice indicating the thoughts of a visible character without the character's lips moving.

volume - The intensity of sound. A volume of 0 is minimum volume. A volume of 100 is maximum volume.

VPI - Video programming interface.

VTR - Videotape recorder.

W

walk-through - A type of animated presentation that simulates a walking tour of a three-dimensional scene.

walk-up-and-use interface - An interface that the target audience should be able to use without having to read manuals or instructions, even if they have never seen the interface.

waveform - (1) A graphic representation of the shape of a wave that indicates its characteristics (such as frequency and amplitude). (2) A digital method of storing and manipulating audio data.

wide shot - Synonym for *long shot* .

wild footage - Synonym for *original footage* .

window - An area of the screen with visible boundaries within which information is displayed. A window can be smaller than or the same size as the screen. Windows can appear to overlap on the screen.

window coordinates - The size and location of a window.

wipe - Technical effect of fading away one screen to reveal another.

workplace - A container that fills the entire screen and holds all of the objects that make up the user interface.

write once/read many (WORM) - Describes an optical disc that once written to, cannot be overwritten. Storage capacity ranges from 400MB to 3.2GB. Present technology allows only one side to be read at a time; to access the other side, the disk must be turned over.

WS - Script abbreviation for *wide shot* .

WYSIWYG - What You See Is What You Get. The appearance of the object is in actual form. For example, a document that looks the same on a display screen as it does when it is printed. Composed views of objects are often WYSIWYG.

X Y Z

Y - Refers to the luminance portion of a Y/C video signal.

Y/C - Color image encoding that separates luminance ((Y) and *chrominance* (C) signals.

YIQ - Image encoding scheme similar to YUV that selects the direction of the two color axes, I and Q, to align with natural images. As an average, the I signal bears much more information than the Q signal. (YIQ is used in the NTSC video standard.)

YUV - Color image encoding scheme that separates luminance (Y) and two color signals: red minus Y (U), and blue minus Y (V). Transmission of YUV can take advantage of the eye's greater sensitivity to luminance detail than color detail.

zoom in - An optical camera change where the camera appears to approach the subject it is shooting.

zooming - The progressive scaling of an image in order to give the visual impression of movement of all or part of a display group toward or away from an observer.

zoom out - An optical camera change where the camera appears to back up from the subject it is shooting.
